AFRL-IF-RS-TR-2005-402
Final Technical Report
December 2005

SEMANTICALLY GROUNDED BRIEFINGS

Teknowledge Corporation

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. K535

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2005-402 has been reviewed and is approved for publication

APPROVED: /s/

WAYNE A. BOSCO
Project Engineer

FOR THE DIRECTOR: /s/

WARREN H. DEBANY, JR., Technical Advisor
Information Grid Division
Information Directorate

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 074-0188*

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503

1. AGENCY USE ONLY (Leave blank)	**2. REPORT DATE** DECEMBER 2005	**3. REPORT TYPE AND DATES COVERED** Final Jul 00 – Oct 05

4. TITLE AND SUBTITLE SEMANTICALLY GROUNDED BRIEFINGS	**5. FUNDING NUMBERS** C - F30602-00-C-0202 PE - 62301E PR - DAML TA - 00 WU - 17
6. AUTHOR(S) Robert Balzer	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Teknowledge Corporation 1800 Embarcadero Road Palo Alto California 94303	**8. PERFORMING ORGANIZATION REPORT NUMBER** N/A
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency 3701 North Fairfax Drive Arlington Virginia 22203-1714; AFRL/IFGA 525 Brooks Road Rome New York 13441-4505	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER** AFRL-IF-RS-TR-2005-402

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Wayne A. Bosco/IFGA/(315) 330-3578/ Wayne.Bosco@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.	**12b. DISTRIBUTION CODE**

13. ABSTRACT *(Maximum 200 Words)*

The vast amount of data contained in the Web hinders the capacity of people to process the information it contains. Teknowledge's Briefing Associate (BA), developed under Defense Advanced Research Projects Agency's (DARPA) DAML (DARPA Agent Markup Language) technical program, facilitates the composition and publication of semantically grounded briefings. BA rendered briefings that contain markups that describe the domain-specific content matter of the briefing. BA briefing may contain either original or imported semantic content. The BA generates DAML descriptions of a briefing's original content as a byproduct of creating that content's visual depiction. Briefing authors' select ontologically defined objects as predefined graphic shapes or icons to include in their briefing mediates the creation of web language markup, for original content. These visually annotated ontologies are demand-loaded into the BA to specialize it to a particular subject-matter domain. BA also graphically generates depictions of imported semantic content. BA is as an extension of Microsoft PowerPoint. The intent of this augmentation is to (nearly) eliminate any cost of producing this semantic markup beyond the costs inherent in producing the equivalent semantics-free version. Teknowledge similarly augmented Microsoft Word to enable authors to add DAML markup to their Word documents.

14. SUBJECT TERMS Semantic Markup, DAML, COTS Extensions, COTS Augmentation, Markup Tools	**15. NUMBER OF PAGES** 64
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

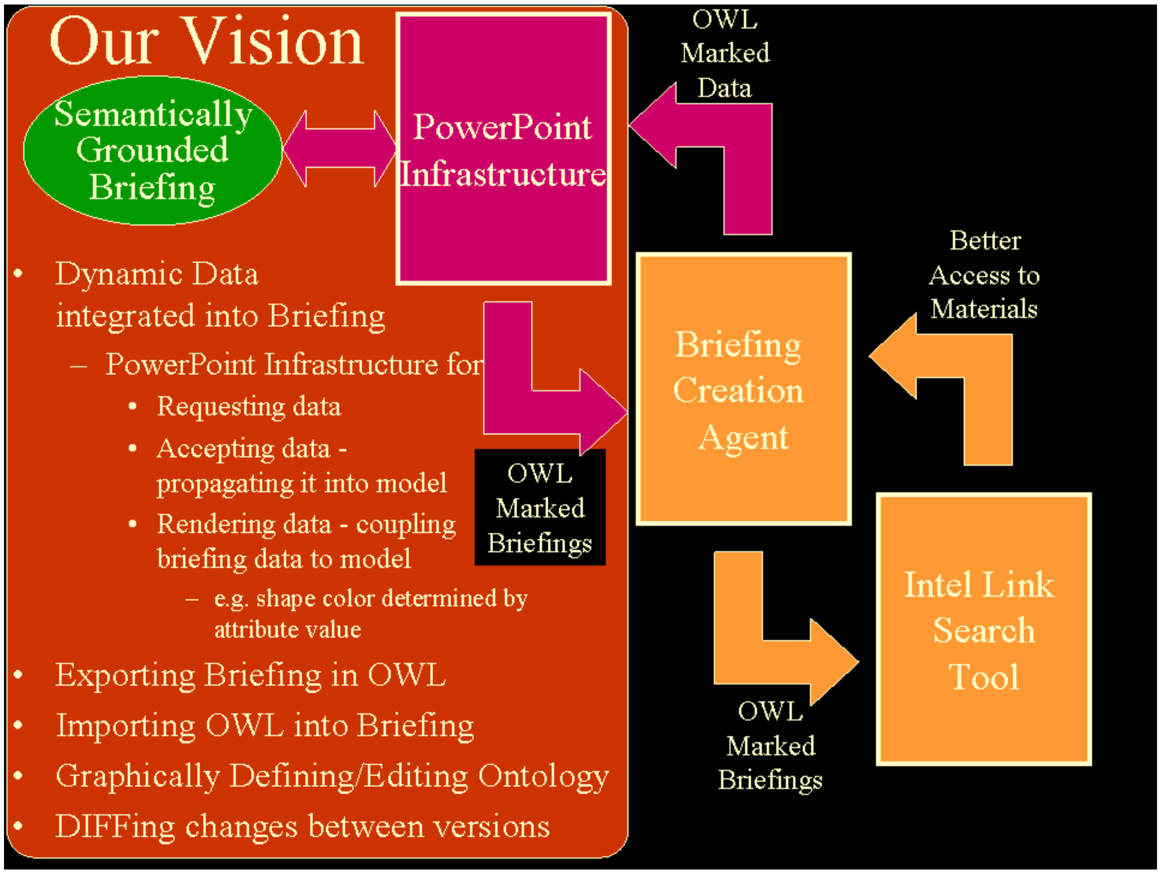
Table of Figures

Table of Tables

1. EXECUTIVE SUMMARY

1.1 Objective

Our main objective was to create a Briefing Associate that generates Defense Advance Research Project Agency (DARPA) Agent Mark-up Language (DAML) or Ontology Web Language (OWL) descriptions of a briefing's semantic content as a byproduct of creating the briefing's visual content so that relevant briefings or relevant portions of briefings can be found by semantic search and to automate the import of semantic content so that briefings could be updated as needed.



- Dynamic Data integrated into Briefing
  - PowerPoint Infrastructure for
    - Requesting data
    - Accepting data - propagating it into model
    - Rendering data - coupling briefing data to model
      - e.g. shape color determined by attribute value
- Exporting Briefing in OWL
- Importing OWL into Briefing
- Graphically Defining/Editing Ontology
- DIFFing changes between versions

Figure 1. Briefing Associate Software Vision

Another objective was matching our markup production capabilities with the needs of the of the intelligence community. The majority of intelligence analyst's use Microsoft Word to produce documents and reports. Teknowledge demonstrated an initial version of a template-based markup product within Microsoft Word with ontologies that could be used used by an analysis to create documents, and perform post-processing to add markup.

1.1.1 PowerPoint and Semantic Markup

PowerPoint (the Commercial Off The Shelf (COTS) tool that briefing authors already use) was augmented to produce briefings with DAML or OWL descriptions as a byproduct of creating the briefing's visual content. These semantic markups are generated as users incorporate shapes or pictures from a visually annotated ontology and connect them into a diagram, and can be tailored for specific subclasses or individual instances.

The intent of this augmentation is to (nearly) eliminate any cost of producing this semantic markup beyond the costs inherent in producing the equivalent semantics-free version.

The visually annotated ontology also provides the basis for automated visualization of imported semantic content. This semantic content, produced by some other DAML or OWL agent can be automatically depicted according to the conventions of the annotated ontology. The author can then manually tailor these depictions and/or augment them with new content, using the ontology-specific editing interface. The Briefing Associate maintains mappings between DAML or OWL descriptions and visual content as the author edits a briefing.

By also retaining a link identifying the source of imported content, the Briefing Associate transforms briefings from information snapshots, whose value declines as the information in those snapshots becomes dated and obsolete, into renewable resources whose information can be automatically updated as needed.

A briefing's content is not limited to material describable by an ontology. The full range of PowerPoint graphics, text, and animation features is accessible through their standard tools. Content not related to a specific ontology is published with markup from a "generic" briefing ontology, which still allows for its discovery by text-based search agents.

The incorporation of analysis and synthesis tools that utilize these semantic markups during briefing composition to improve the resulting document's accuracy, quality, and/or speed of production was also explored.

1.1.2 Microsoft Word and Semantic Markup

The HORUS project, a DARPA Technical Integration Experiment (TIE3), required the deployment of MS-Word based markup tool that:

- Automatically markup an intelligence type report document using Lockheed's Aero DAML natural language analysis software.

Our project provided the means for seamless transmission of a word document's content to an AeroDAML server and incorporation of the results of that analysis into the source Word document.

- Insert pre-composed templates into the document to create a summary.

The templates contain natural language text that is annotated with DAML markup. Key phrases in these templates are marked as "variables" and annotated with nothing but a class and/or co-reference restrictions with other variables.

- Instantiate author inputted template variables.

This is takes place primarily through drag/drop from the automatically produced markup onto the variable slots of the templates, although the implementation also allows instantiation (and extension) of the templates with "manual" markup produced by Graphical User Interface (GUI) tools specialized to the governing (HORUS) ontologies.

1.2 Accomplishments

1.2.1 PowerPoint based Briefing Associate

All accomplishments are available as augmentations within PowerPoint.

- *Ontology-Aware Briefing Editor*: Allows a briefing author to create original graphic content that can be automatically annotated with the markups contained in the ontology from which the content was selected.
- *Ontology Importer*: Allows externally defined DAML or OWL ontologies to be used as basis for creating briefings.
- *Visual-Annotation Ontology Editor*: Allows (imported) DAML or OWL ontologies to be augmented with the visual representation of ontological concepts. Creates instantiation tools enabling authors to incorporate instances of ontological concepts in their briefing.
- (Domain Specific) Semantic Content Import and Update: Allows externally produced DAML or OWL content to be imported into a briefing and graphically rendered. Only works for specific domains.
- Enhanced Graphic Rendering: Enhance the graphic rendering of DAML or OWL objects so that the appearance of the object will dynamically change to reflect the values of (some of) its properties.
- DAML Query Interface: Provide a facility for users to specify DAML-based queries and import the object(s) satisfying those queries into the briefing.

1.2.2 Word based Briefing Associate

We demonstrated an extension to Microsoft Word that provides an approximation to the Briefing Associates paradigm of creating markup as a byproduct of document authoring. The demonstration consisted of:

- An MS Word addin to maintain, internally, an association between contiguous text ranges and markup
- Integration of MS Word with AeroDAML, a web-service developed by a group led by Paul Kogut of Lockheed. AeroDAML uses Lockheed's AeroText natural language analysis software to produce markup text submitted to the server. Our integration took advantage of the *smart tags* technology of Office 2000 to automatically invoke AeroDAML in the background on paragraph-sized text chunks as the user edits a Word document. The returned markup is associated with the text, and marked internally as *hypothesized* markup.
- Integration of MS Word with OntoMat. OntoMat is a Java-implemented GUI for manually creating Resource Description Framework (RDF) markup. OntoMat was

developed by Siegfried Handschuh at University of Karlsruhe and Stanford University, for adding markup to existing web pages. Our integration allows the user to create, view, and edit markup associated with regions of text in a MS Word document. Markup created through the OntoMat interface is marked internally as *confirmed* markup.

- Extension of the ontology annotation notation used by the Briefing Associate with *text templates*. These annotations use BA MS Word extension to format (color, font, size, face) text that has associated markup. The annotation-driven formatting can be toggled on or off at the author's discretion.
- Wrapping Word's cut/copy/paste operations so that markup associated with text is retained across these operations.
- Extension of the MS Word GUI with an ontology-specific toolbar, exactly as was done for the Briefing Associate (the two extensions share the same binary code). This toolbar provides a rapid means of adding *class* membership annotations to regions of text. The Word GUI was also extended to provide *context menu* access to a dialog for editing datatype properties in markup, also sharing binary implementation with the Briefing Associate.
- Persistence of all the markup associated with a Word document as part of the saved word ".doc" file.

1.2.3 Ontology compiler

We produced an automatic generator of O-O programming libraries (.Net) from Ontologies. Applications that read/write models based on an ontology are far easier to program through the classes of these libraries than through the classes of ontology-neutral libraries such as Jena.

1.2.4 Internet Explorer Plug-in

We demonstrated a preliminary version of our DAML markup plug-in for Microsoft's Internet Explorer. Besides providing semantic markup capability to a much larger user community, this effort has greatly accelerated our development of markup infrastructure by identifying common components that can be shared among our COTS platforms.

1.3 Technology Transition

The implementation of the Briefing Associate was as an extension of Microsoft PowerPoint so that user, briefing authors, required neither computer programming skills nor Semantic Markup expertise to create and edit semantically grounded briefings. The military and the intelligence community could easily adopt the Briefing Associate. Moreover, the military's extensive reliance on time and mission critical briefings makes the Briefing Associate a potential "killer-app" for the military. In fact, the widespread use of briefings and the ease of understanding the value-added of making them semantically searchable, reusable, and updateable on demand will enable the Briefing Associate to showcase the benefits derivable from the DAML program.

The Briefing Associate's extensions were implemented in terms of published and stable programmatic interfaces to PowerPoint, its implementation will port easily to future commercial versions of PowerPoint.

A collaborative effort with the HORUS project was to make the BA's semantic markup capability directly available to the intelligence analyst community.

1.4 Deliverables

Our software deliverables were plugins for commercial desktop applications (MS PowerPoint, MS Word). Both are deployed in the form of Windows NT/2000/XP installation modules (.msi files). Downloadable copies exist on the Teknowledge web server.

2. POWERPOINT BASED BRIEFING ASSOCIATE

The Briefing Associate (BA) facilitates the composition and publication of semantically grounded briefings. The briefings contain markups that describe the domain-specific content matter of the briefing and are linked at a fine granularity to units of visual content in the briefing. A briefing may contain both original and imported semantic content. The BA generates OWL (or DAML) descriptions of a briefing's original content as a byproduct of creating that content's visual depiction. Visually Annotated Ontologies (VAO) from which authors select ontologically defined objects as predefined graphic shapes or icons to include in their briefing mediates the creation of OWL markup, for original content. These visually annotated ontologies are demand-loaded into the BA to specialize it to a particular subject-matter domain. They also permit the BA to generate graphical depictions of imported semantic content.

The BA, implemented as an extension of Microsoft PowerPoint, allows briefing authors familiar with that product to rely on the native user interface tools, menus, and direct-manipulation actions to edit visual content. Extended interpretation of these tools and actions, and additional tools created from the ontology annotations, simplify the creation of new content, while simultaneously creating OWL markup. Figure 2 depicts the BA's architecture and major information flows.

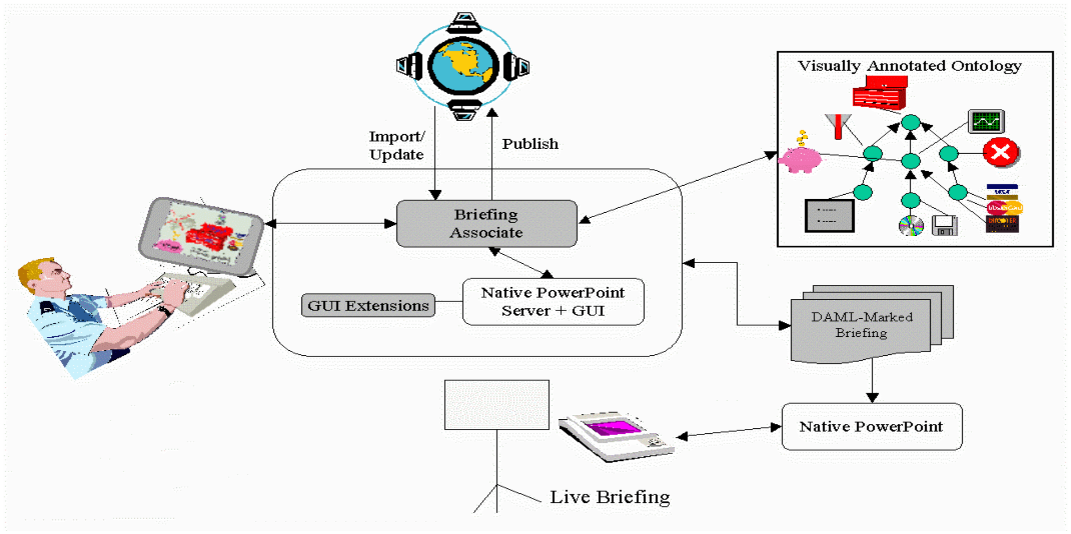


Figure 2. Briefing Associate Software Architecture

The Briefing Associate augments PowerPoint with graphics-bearing ontological categories. These graphics represent instances of domain concepts, attributes (primitive data typed properties), and their relationships. The author, while composing a briefing using these graphics is indirectly constructing a semantic description of the briefing content. Besides supporting the construction of semantically grounded briefings, the Briefing Associate also exposes the briefing's emerging semantic descriptions to external modules called analyzers that perform specialized services or analyzes for the author. These analyses can provide feedback to the author, can extend or modify the briefing, or can produce external documents. One particular generic analyzer is a publisher that generates the semantic markups that describe the briefing content. The Briefing Associate extends the PowerPoint GUI with tools, menus, and gestures for instantiating the semantically annotated graphics, assigning attribute values to the instances and relations represented by these graphics, invoking analyses, importing and updating the graphic representations of imported semantic descriptions, and annotating domain ontology concepts and relations with their visual representations. The following subsections describe the components that achieve these added services.

2.1 Ontology Aware Briefing Editor

The Ontology-Aware Briefing Editor allows a briefing author to create original content and to import and edit externally produced content. Visually annotated ontologies, discussed below, provide the means to relate OWL descriptions from a given ontology to a visual model.

Ontology-aware editing takes place through a combination of standard PowerPoint interface actions, additional GUI elements added by the BA, and extended interpretation of native controls and direct-manipulation actions. The entire native PowerPoint user interface continues to be functional. User-preference tailoring of that interface is preserved.

A visually annotated ontology is the key to creating original content as well as to automatically depicting imported content. A new toolbar is added to the PowerPoint GUI for each referenced ontology. For each concept and relation in an ontology, an instantiation tool is added to the toolbar. Our current implementation lays these tools out in a single list. We also plan to offer these tools in a cascading interface, mirroring the class inheritance of the ontologies. Clicking on one of these tools, like PowerPoint's native autoshape tools, allows the author to insert a copy of the graphic template anywhere in his briefing. Domain relations defined in the Ontologies are graphically depicted by arrows (more precisely PowerPoint connectors) whose ends are attached to the concept instances related by that relation. These instantiation tools simultaneously create the internal semantic representation for that concept or relation instance as defined by the ontology (including any default attribute values).

To accommodate briefings that utilize classes from multiple ontologies, the user interface displays a separate toolbar for each basis ontology. Exactly one toolbar is created for each ontology used by any briefing loaded during a session; that toolbar is shared by all briefings having the ontology as one of its bases. When a briefing is activated, its toolbars are enabled and made visible. Initially, the toolbars are docked at the top of the PowerPoint GUI. However, as standard Microsoft Office toolbars they can be repositioned or floated at the user's option.

Ontology-independent tools and predefined menu options reside on a distinct toolbar, which is visible and enabled whenever the Briefing Associate itself is enabled. This toolbar includes the tool for assigning an ontology to a briefing.

The Ontology-aware briefing editor also allows the author to edit domain attribute values through a dialog box interface that is activated from the context menu of the graphic representing that instance in the briefing. A tabbed dialog is created for the selected instance with a tab for each attribute applicable to that instance. Each tab provides an interface, specific to the attribute type, for viewing and setting the value of that attribute.

Figure 3 is a screen shot of the ontology-aware briefing editor in a "satellite communications" domain. Everything in the figure is part of the GUI with the exception of the callouts highlighting specific elements.

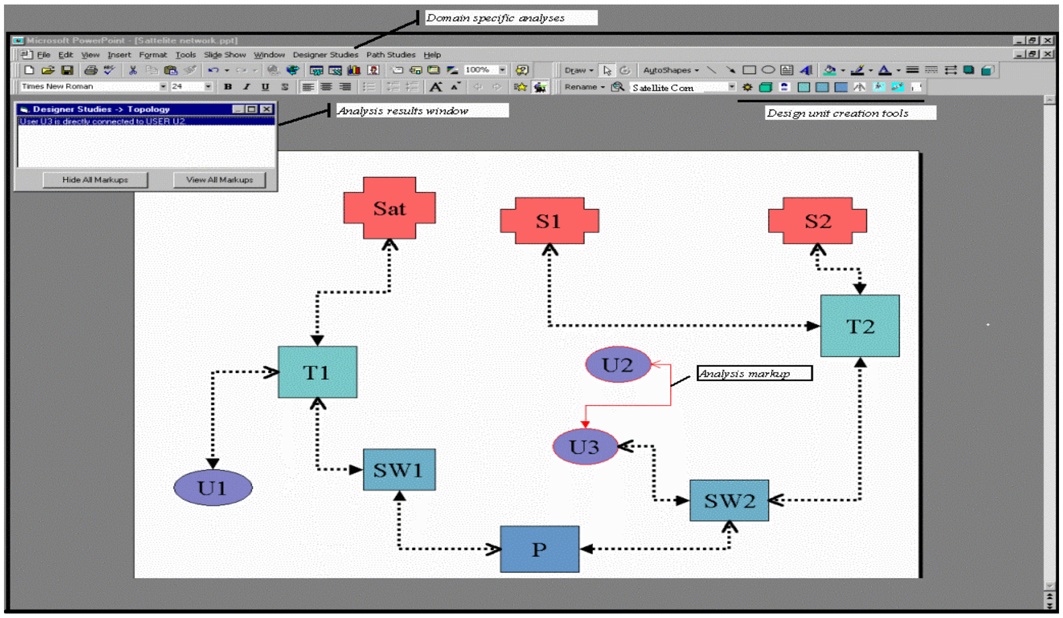

User U3 is directly connected to USER U2
Domain specific analyses
Microsoft PowerPoint - [Sattelite network.ppt]
File Edit View Insert Format Tools Slide Show Window Designer Studies Path Studies Help
Times New Roman
24
100%
Draw
AutoShapes
Rename
Satellite Com
Designer Studies -> Topology
Hide All Markups
View All Markups
Analysis results window
Design unit creation tools
Sat
S1
S2
T1
T2
U2
Analysis markup
U3
U1
SW1
SW2
P

Figure 3. Ontology aware briefing editor GUI – satellite communications

In the central canvas is the depiction of a "satellite communications" configuration. The various labeled shapes represent instances of satellites, terminals, switches, processors, and users – the domain concepts defined in the ontology. They are connected by arrows representing communication links – the only (non data-typed) domain relation in this ontology.

The author created the preponderance of this briefing through the instantiation tools on the domain toolbar, on the right side of the second row of docked toolbars at the top of the figure. To the immediate left of these instantiation tools in the domain toolbar is a drop-down list box displaying the name of the current ontology ("Satellite Com"). When a briefing author starts a new briefing, this box allows him to choose an ontology. This triggers the creation and display of the appropriate domain toolbars for that ontology. Manipulation of the concept and property instances on the briefing – positioning, resizing, selecting, attaching/detaching links– is carried out through PowerPoint's native mouse gestures and/or keyboard shortcuts.

In Figure 2 the user has requested a "topology" analysis, one of the analyses in the "Designer Studies" group. The results of this analysis are displayed in a separate window, visible at the upper left of the canvas in Figure 2. The window displays a list of reports. In this example, there was just one report. Its explanation reads, "User U3 is directly connected to user U2." When the user selects one of these reports, its associated markups are displayed as highlights. In this case, the only markup called for highlighting the communication link between U2 and U3. That is why that link has an appearance (a thin red arrow) different from the others. The effect of this highlighting is reversed when the report is deselected or the analysis window is closed.

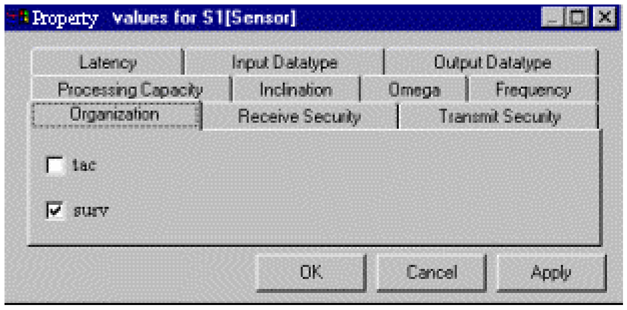


Figure 4. Property value dialog

Attribute values are viewed and assigned through dialogs, displayed on demand from the graphic instance's context menus. Figure 4 exhibits the dialog for a sensor satellite. The dialog contains a "tab" for each attribute associated with that concept in the ontology. The details of a tab depend on the value type of the attribute and on the concept specification.

Identical dialogs are used to gather the parameter values for parameterized analyses.

When PowerPoint is put in "slideshow" mode – the generally preferred mode for briefing presentation, but not development -- its user interface changes, both in appearance and in response to mouse and keyboard input. Heretofore, a briefing produced with the Briefing Associate could be shown as a slideshow, but the Briefing Associate's own GUI enhancements were inaccessible from the slideshow presentation.

We enhanced the Briefing Associate's handling of PowerPoint events so that a briefing presented as a slideshow still allows the presenter access to:

- Analyses information, including the selective display of visual feedback from analysis reports

The analysis report window can be set to "float" on top of the full-screen slides. This floating option is also available in development mode, and can be toggled on/off in each individual analysis windows. In slideshow mode, floating mode is the default.

- The dialogs that display values of datatype properties associated with class instances

2.2 Visually Annotated Ontology Editor

The Briefing Associate is not limited to any particular ontology; any OWL ontology can be annotated. Multiple annotated versions of a single ontology may be created, so that briefings can be tailored easily to different audiences with different conventions for the visual representation of information. However, any single briefing will be based on a single visually annotated ontology.

The Visual-Annotation Ontology Editor provides an interactive means to establish a mapping between the concepts of an ontology and their visual representation. When an ontology O is imported into the VAO editor, the editor lays out O's concepts and properties depicting their hierarchical relationships (i.e., the subclass and sub property properties). The user can assign graphic representations to these concepts and properties and assign icons to be used in the ontology tool bar used for briefings to be associated with O. The user also indicates the analyses that briefing authors will be able to invoke from the ontology-aware briefing editor through its GUI.

The VAO editor is actually a specialization of the ontology-aware briefing editor that uses the visual annotations defined for the ontology domain. These visual annotations allow the object and relation types in an ontology to be defined graphically. These ontology annotations are thus just briefings in this ontology domain and are saved as a PowerPoint presentation. They are loaded on demand by the Semantic Content Import and Update and Ontology-Aware Briefing Editor components. Importing an ontology is thus a case of content import, while adding visual annotations is a case of original content creation.

Figure 5 shows the Satellite Communications visually annotated ontology used in the example of Figure 3. The (green) rectangles labeled "Comsat", "Sensor", "User", etc. represent the leaf domain concepts. The cross shapes attached to them by dashed connections are their graphic templates. This determines the appearance of an instance of that concept. Any of PowerPoint's native autoshapes, formatted as desired, may be used as a graphic template. Alternatively, an image may be chosen as a graphic template.

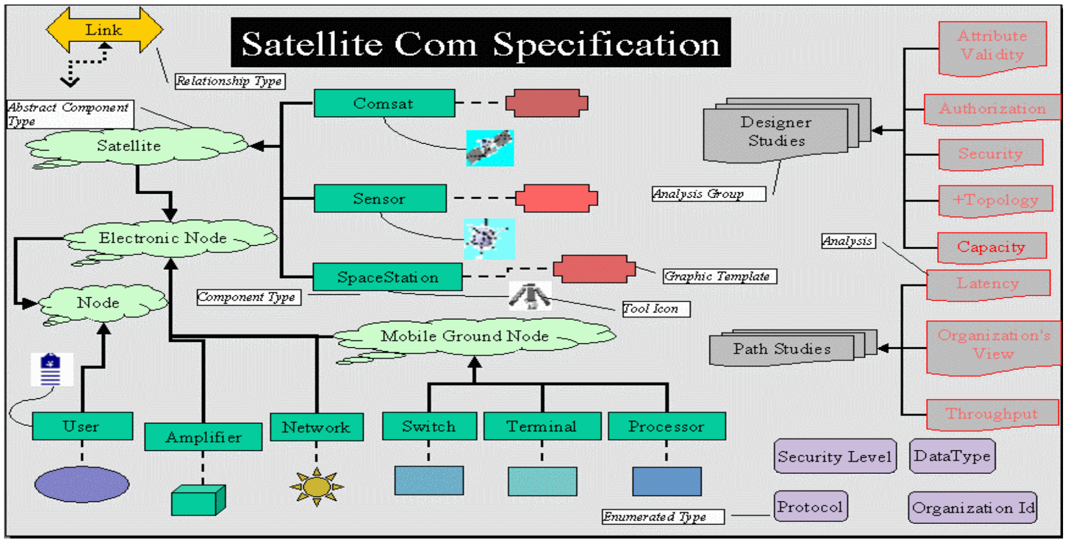


Figure 5. Visually annotated ontology – satellite communication

A concept may be connected (via a curved solid connector) to an image that serves as the tool icon for that concept in the instantiation toolbar. Tool icons, like graphic templates, may be selected from a shape library or may be imported. If no tool icon is specified, a scaled version of the graphic template is used as the tool icon.

The (light green) clouds labeled "Satellite", etc., represent the non-leaf concepts. The (gold) arrow shape labeled "Link" defines the sole relationship type in this domain. The dashed, double-headed arrow attached to it is the graphic template for the "Link" relationship type. The user tailors the color, dashing and arrowhead styles of a relationship template in the graphic domain specification just as he tailors component type templates.

Any concept or relationship type may have initial attribute values specified through a property-editing dialog, identical to the ones used by the ontology-aware briefing editor. The default values are assigned when new instances of the type are created.

Figure 5 contains the specification of two analysis groups, “Designer Studies” and “Path Studies”, and eight analyses in those analysis groups. The color and styling of the border of an analysis specify the means used to highlight components and relationships identified in reports in the feedback from the corresponding analyzers. For instance, the “U2-U3” connection in Figure 3 was highlighted as a thin red line because the border of the “Topology” analysis is a thin red line. Analogously, the text characteristics – font, face, size, color – of the label of an analysis specify the textual characteristics of any markup text found in feedback from the analysis.

Extending Ontology Definitions

The Briefing Associate evolved from a research system that predated the OWL (or DAML) language. It had its own set of “description logic” primitives, resembling the primitives of a traditional object-oriented language. Although we had mapped these to OWL for the purpose of publishing a briefing’s content, it was not possible to map a typical OWL ontology onto the existing Briefing Associate primitives because OWL is able to express more – and in a few cases, less – than the declarations of an Object-Oriented (OO) language. We have now extended the Briefing Associate’s “ontology” domain to be in near 1-1 correspondence with OWL, including:

- Multiple inheritance
- Sub classing of instantiable classes (not just interfaces)
- Expressing (unconditional) domain and range class constraints on relations
- Defining classes using union/intersection/complement operators
- Mapping primitive value types to XML Schema datatypes
- Including a slot for a URI on both the internal representation of ontology concepts and on individuals in briefings

Two fundamental issues remain in making the Briefing Associate “complete” with respect to the OWL language. Both have to do with the fact that the Briefing Associate’s ability to support efficient, and even incremental, analysis of briefing content relies on mapping the content of a briefing into an object model. This in turn relies on mapping the classes, properties, and restrictions of an OWL ontology into the class and property declarations required to support such a model. The two difficulties are:

A description logic language, such as OWL, permits one to create descriptions that are *inconsistent* with the ontology on which the description is based. E.g., one can describe a person with two distinct birth dates even if the ontology claims birth date to be a unique property. The implementations used for OO languages, however, generally take the restrictions imposed by class and property declarations into account as part of their representation of a model, and hence preclude the creation of models that are inconsistent with those declarations.

A description logic’s ontology language, such as OWLs, permits one to define classes that are neither declared to be *disjoint* nor declared to bear any subclass relation to one another. This means that, for any such set of classes, a perfectly consistent description may state that an individual belongs to all the classes in the set. OO language implementation techniques, however, generally restrict an individual to membership in a single (named) class and its super classes, and require the named classes to be declared *prior* to the construction of specific models. Thus, e.g., one cannot create an object that is both a *military officer* and a *head-of-state* unless one has anticipated the need for the intersection of those two classes in advance. But it is impractical to compile OO libraries in which all possible subsets of independent classes have been reified through multiple inheritances.

We note that completeness, in the sense just described, is *not* a necessary attribute of a useful tool for creating markup. Although it is not essential for consuming markup, either, a useful tool should at least be able to explain to a user where it is unable to represent the content being consumed.

2.2.1 Briefing Associate Graphic Modulators

The visual annotation language for the Briefing Associate was extended so that the value of a datatype property of an individual can influence the graphic rendering of that individual in a presentation. Heretofore the value could only be displayed as text in a text-bearing shape that was part of the graphic template for the individual.

A *Graphic Modulator* (GM) for a class C consists of:

- A *modulating condition*, which consists of a single datatype property P for which instances of C can serve as the domain, one or more values from P's range (or an open or closed range of values in the case that the range of P is numeric). The condition can be either that *some* value of P or *all* values of P be among the identified range values.
- A *modulating graphic*, which can be any PowerPoint autoshape, group shape, or image
- Identification of a *modulated component* of C's graphic template. If C's graphic template is not a group shape, then the modulated component is the entire template.
- Specification of the *modulated features* of the modulated component. The modulated features can be any subset of PowerPoint's graphic properties (e.g., fill and line colors, line style and weight, font characteristics) or the pseudo-feature "replace"

Whenever the modulating condition of a GM holds for an individual I belonging to GM's class in some model, the renderings of its proxies in the model are modified. If GM uses the pseudo-feature "replace", then a copy of GM's modulating graphic replaces the modulated component of each proxy. Otherwise, the graphic property values of GM's modulated features are copied from the modulating graphic to the modulated component of each proxy.

The implementation does not restrict the number of GMs that may be specified for a given class. Multiple GM's for a class may use the same property in their modulating condition.

When the conditions of multiple GM's for a class hold for a single individual, *all* the modulation is applied. If two or more of these GM's share the same modulated component and have any overlap in modulated features, the resulting feature values are implementation-dependent. However, it is guaranteed that if any of the GMs for a component uses the pseudo-feature "replace", then exactly one of those, and none of the others, will be applied.

The GUI for graphic modulator annotations consists solely of graphics and dialogs. No programming is required. A single modulating graphic can be shared by multiple graphic modulators. In particular, a set of graphic modulators that apply to the same modulated component but whose conditions are based on different datatype properties can be specified very concisely.

The Briefing Associate applies and removes graphic modulation dynamically as a modeler changes the values of datatype properties in a model.

2.2.2 Property and Parameter Values

Among the annotations of a Visually Annotated Ontology are ones that specify initial values for:

- Datatype properties of newly-created individuals, based on the class of the individual
- Datatype parameters of analyses, based either on the identity of the analysis or the analysis group to which it belongs.

We learned that it is sometimes convenient to be able to specify these initial values in terms of some aspect of the presentation or its environment, rather than as a literal value. However, we did not want to introduce much complexity into our annotations. As a compromise, the strings used to specify initial values now undergo *environment variable* substitution. When initializing datatype parameters of analyses, the Briefing Associate augments the operating system's environment variable set with:

DesignName –the file name of the briefing

DesignFolder—the path to the folder in which the briefing is saved

Analyses are often run repeatedly, and it is inconvenient to have to reenter parameters that do not default to the desired value. In the past, the Briefing Associate kept a session-duration cache of parameter values for each analysis. Whenever an analysis was invoked, the cache for that analysis, if present, was used to provide initial values for the parameters. We refined this so that an independent cache is kept for each [briefing, analysis] pair. Thus, the initial parameter values for an analysis are the ones used most recently when applying that analysis to the *same* briefing.

2.3 Analyzers and Applications

Analyzers are external executable modules that process the internal semantic descriptions of the briefing content to provide an analysis, a synthesis, or some other service. An analyzer can be implemented so as to execute within the PowerPoint process, as a separate process on the same machine, or (via Distributed Component Object Model (DCOM)) on a different workstation. Analyses are associated with a particular domain and this association is indicated in the VAO and they are invoked through the BA editor menu for that domain. When an author requests an analysis, the BA creates a connection to the module implementing that analysis and passes it a reference to the briefing to be analyzed, together with any author-provided parameters for the analysis. That analyzer is subsequently expected to send the BA a set of reports describing the analysis performed, the synthesis done, or the service rendered. The BA then presents the report(s) to the author.

For a snapshot analysis, the analyzer's responsibility ends with transmission of the reports detailing that analysis. An incremental analysis, however, is expected to send updates to its reports as the author continues to modify the briefing, until the author closes either the analysis or the briefing. To support incremental analyses, the briefing reference handed to the analyzer by the BA provides not only direct access to the content of the briefing, but to events representing changes to that content.

A transaction grouping is imposed on top of events. It is these transactions, not primitive events, that represent the unit of change to which an incremental analyzer commits to respond with updated analysis reports. Because the responses are permitted to be asynchronous, they are accompanied by the transaction id of the transaction that triggered them. This allows the BA to understand, and reflect in its GUI, whether a displayed set of analysis reports is up-to-date.

Although the briefing reference provided to an analyzer can be used to gain direct access to PowerPoint's detailed graphic model of a briefing, analyzers are typically interested in the ontology-based model information that is being automatically generated when content is imported from the semantic web or created through tools associated with the VAO. For each ontology, a .NET library called an *Exchanger* is automatically generated. .NET is Microsoft operating system platform that incorporates applications, a suite of tools and services and a change in the infrastructure of the company's Web strategy This library reflects a straightforward mapping between classes and properties of the ontologies and the corresponding modelling concepts (classes, interfaces, and properties) of .NET. Most, if not all, widely-used programming language *Integrated Development Environments* (IDE) for the Windows platform provide a declarative way to import such a library, automatically building the client-side code needed to program directly in terms of the objects exposed by the library.

The code generator for Briefing Associate exchangers, and the runtime for the Briefing Associate GUI and analyzers, were extended so that *all* non-graphic information about an ontology that is required for building or analyzing models based on that ontology resides in the ontology's exchanger. Previously, some of this information was obtained from the ontology's annotations presentation.

This change, in conjunction with the ability to generate exchangers directly from the Extensible Markup Language (XML) representation of OWL ontologies (see Ontology Compiler), means that it will no longer be *necessary* to define and maintain an ontology using the Briefing Associate's own (PowerPoint) GUI in order to use that ontology as the basis for models built, maintained, and analyzed from that GUI. Annotation files will need only to include the graphic annotations to an ontology, not the information – such as hierarchy of classes or domains and ranges of properties – standardized by OWL itself.

2.3.1 Generated Generic Publisher (original approach)

To complement our already implemented means of *persisting* the semantic model as part of the PowerPoint presentation, we provided a means to independently save a persistent representation of the semantic content in the form of OWL markup, using the approved American Standard Code for Information Interchange (ASCII) XML encoding of OWL.

We refer to the code that implements persisting of the semantic model as the briefing *publisher*. Since the publisher meets the requirements of a Briefing Associate Analyzer, it can be invoked at any time from within the Briefing Associate via a PowerPoint menu item. Since we actually *generate* a publisher for each ontology, there is technically an open-ended set of publishers. The generator is itself an analyzer, registered for the Ontology Definition domain. It can be invoked from the Briefing Associate while editing an ontology. The generated publisher accepts two parameters:

- The file where the OWL is to be written
- A namespace for the Uniform Resource Identifiers (URI) created to represent the individuals in the briefing.

The publisher is generated as Visual Basic (VB) source code. If the VB development environment is accessible to the Briefing Associate at the time the code is generated, the VB compiler is invoked automatically to produce the binary form of the publisher, and it is registered as an analyzer for the ontology.

At the suggestion of the DAML program manager, we provided an option for augmenting the semantic content with string-valued properties containing strings appearing in a briefings graphics. Three distinct attributes are used to identify titles, subtitles, and shape labels. The *meta* data that PowerPoint maintains about each presentation – author name, company, creation date, etc – can also be include in the OWL output from the publisher.

2.3.2 Interpreted Generic Publisher (Final Approach)

Heretofore the Briefing Associate persisted presentations via an analyzer that needed to be individually generated for each ontology. This design had two major drawbacks:

- Any change to an ontology required *regeneration* of a publisher for that ontology. It was necessary to manage consistency between versions of ontology definitions and the corresponding publishers.
- Changes to the Briefing Associate's modeling choices could necessitate changes to the publisher generator. In that case, *all* publishers would need to be regenerated. Changes to the OWL language, or bugs discovered in the publisher generator, would induce the same necessity.

Using the added *reflection* capabilities, we implemented an ontology-independent publisher for the Briefing Associate. The new publisher works by interpreting the reflection information available for a briefing's ontology, and gathering data from the briefing's exchanger based on that information. This eliminates the need to generate a specific publisher for each ontology, and ensures that the content published for a briefing is consistent with the version of the ontology in use at the time of publication. Because publication is performed infrequently (not as part of each edit to a briefing), the additional cost entailed by an interpreted algorithm is acceptable. The new publisher is accessible from *any* briefing, via a menu item on the Briefing Associate toolbar.

2.3.3 Ontology Importation

We wrote an *analyzer* for the Briefing Associate's *ontology* domain that allows a user to import an existing OWL ontology into the Briefing Associate. This analyzer is invoked from a (normally empty) ontology definition "briefing", accepting the name of a file containing (the XML encoding) of a OWL ontology as input. The analyzer adds instances of the classes and properties defined in the OWL ontology to the briefing. The user (an ontology designer) can then add the visual annotations that are needed to make an ontology useful as the basis for briefings containing markup constructed from the ontology's concepts.

This analyzer uses some simple partitioning and layout heuristics to factor the visualization of the ontology across multiple slides. The user will generally choose to adjust this partitioning and layout. This task is simple using the standard PowerPoint graphic editing tools, especially given the fact that the Briefing Associate preserves its understanding of the concept behind a graphic across cut/copy/paste operations.

The ontology importer relies on the Briefing Associate's *proxy* capabilities – multiple graphic instances known to represent the identical individual – to avoid overly intricate visualizations. For example, a class's *properties* are visualized on a different slide from its *sub/super-classes*.

One significant lesson learned from work on the ontology importer was that a direct translation of an ontology into the Briefing Associate's primitives will often produce a sub-optimal internal model. The primary problem is that a direct translation will *never* employ the Briefing Associate's ability to apply attributes to *instances* of properties, because OWL does not consider *triples* to be individuals in their own right.

In our implementation of the ontology importer, we dealt with this problem by supplying the analyzer with a second input, which was a textually-defined set of *ontology mappings*. The effects of these mappings are reversed when a briefing created with the imported ontology is published, so that the published version conforms to the original form of the ontology. However, we believe that composing these textual ontology mappings is error-prone, and that the same effect would be better accomplished by post-importing transformations initiated through the Briefing Associate's GUI.

2.3.4 Visual Queries

Our first step toward supporting *content import* (distinct from *ontology import*) was to create a means for composing *visual* queries. We chose to adopt a very simple query convention. A quantification-free form of query can be considered to be simply an OWL *description* in which some individuals are considered *typed variables*. The *answers* to the query, vis-à-vis some existing set of OWL triples, is a set of bindings of the variables to URIs of individuals from the triples, such that the instantiated description would be entailed by the set of triples. Identification of the *variables* in our interface is accomplished by selecting (in the PowerPoint GUI) one or more individuals in the description.

A richer query language can be obtained by adding disjunction and negation to the description. We added visual units for these logical primitives. Although conjunction is really implicit in existing OWL tools – a set of triples in a file is considered to represent the conjunction of their assertions – we also needed to add a *conjunction* primitive to our visual query language for flexibility.

We built a very simple query engine that would suffice for small databases of triples, and allow us to demonstrate the concept of visual query.

2.3.5 Application: Image Annotation

To demonstrate content import via visual query, and simultaneously demonstrate an interesting form of legacy data markup:

- We created a simple ontology allowing descriptions of paintings, artists, and museums, and added visual annotations.
- With this visually annotated ontology, we used the Briefing Associate to create descriptions (in the form of a briefing) of a few dozen actual paintings. In our briefings, we included images of these paintings, downloaded from web servers. Our descriptions contained the URLs of the pages holding these images as properties of the paintings.
- The Briefing Associate published the briefing's content as OWL markup, creating a database for us to pose queries against.

Since our query engine simply produced URIs of individuals in the database, we still needed a way to present the results to a user – simply presenting the URI as a string would not be helpful. Here, we recognize that a visual query really needs to include some notion of a *view* of the result. We have not yet implemented that notion. For the demonstration, we limited ourselves to queries involving a single "variable", of type painting (or a subclass), and presented the results via a procedurally encoded view that displayed the painting, its painter and his/her nationality, and the museum displaying the painting (if known). The manually encoded view displayed each result, using the graphical conventions of the visually annotated ontology, on a distinct slide added to the "briefing" from which the query was posted. We also provided a "context menu" option that allowed a user to visit, in Internet Explorer, the web page identified by the URL (if any) associated with a painting. The *layout* of the graphics within a slide in the result is determined by heuristics in the view.

Because the presentation of the answer to a query is itself valid Briefing Associate content, it can be further edited – e.g., to add properties not known to the database or not presented in the view – or used as the basis for further queries.

The association between paintings and URLs was actually indirect, via a property that contained a *digital signature* of the image. The effect of this was that, at some later time, our "database" could be augmented by a web crawler that found images of paintings and associated the URLs with the digital signature of the image. With no further manual markup, if one of these additional URLs U held an image identical in signature to that of the image from a URL V associated with an already marked-up painting P, then U as well as V would be known as a web location where a painting of P could be found.

2.4 Implementation

The Briefing Associate is a descendent of the Design Editor, an application for producing visual domain-specific design environments. The Briefing Associate, like the Design Editor, is implemented as an extension of Microsoft PowerPoint. We regard this choice not as an implementation detail, but as central to this research. First, PowerPoint provides us as implementers with a far higher-level platform for building a briefing tool than generic middleware, such as Component Object Model. (COM)/ Common Object Request Broker Architecture (CORBA) and GUI widget libraries. It provides an extensive ontology for representing the visual content of briefings, and support for making models that use that ontology persistent. Furthermore, it provides an extensive What You See Is What You Get (WYSIWYG) user interface for viewing and editing the visual content of a briefing. This interface requires some extension, but no redesign or reimplementation, to accommodate OWL-aware briefings. Second, PowerPoint is the most widely used product for authoring briefings and hence it facilitates the adoption of the BA by briefing authors.

The BA is programmed primarily in Visual Basic. For PowerPoint, this extension is a COM addin that receives "events" as the user creates, opens, closed, and modifies briefings. As a client of PowerPoint, this module can navigate through a briefing and paint analysis feedback directly onto it. For efficiency reasons, this module runs entirely as an "in-process" component. This means it is incorporated into the PowerPoint process itself. Method calls are extremely efficient when both client and server are part of a single operating system process. Greater efficiency could be achieved by implementing the BA in C++, but the performance of the Visual Basic code has been acceptable to date.

PowerPoint's native extension mechanisms include a general, albeit low-level, ability to add arbitrary non-graphic information to a presentation and retain that information in the presentation's persistent file format. The BA implementation relies on this mechanism to retain all ontology-related information about a presentation across editing sessions – it does not attempt to infer ontological information on the basis of graphic attributes of existing graphic objects.

2.4.1 .Net implementation Upgrade

The implementation of the Briefing Associate was upgraded from Visual Basic 6 to Visual Basic.Net, so that it relies on the Microsoft.Net runtime environment rather than the far more limited Visual Basic 6 (VB6) and COM runtime libraries. Among the benefits of the .Net environment for the Briefing Associate are:

The availability of *class inheritance* significantly increased the amount of Exchanger implementation that could be shared across ontologies. The shared code resides in classes that are part of the Briefing Associate itself, and is invoked from methods of subclasses in the ontology-specific exchangers. The resulting exchangers are less than 20% the size of the older ones.

True garbage collection, rather than the reference counting method of memory management used by COM, eliminates the need for carefully crafted code to break reference cycles.

Version management with .Net assemblies is much less problematic than with COM clients/servers. This is particularly valuable because the Briefing Associate's core software component, developed by Teknowledge, must be kept consistent with exchangers and analyzers created (or at least compiled) by users. The .Net runtime is much better able to determine whether an exchanger or analyzer needs to be recompiled in the face of a newer version of the core software.

The .Net runtime environment includes the ability to "compile" source code into "intermediate level code (ILC)". .Net assemblies contain ILC, which is compiled into native machine code when it is loaded into a process. As a result, it will no longer be necessary to have access to a Visual Basic development environment (a licensed product) in order to compile an Exchanger generated by the Briefing Associate for a new ontology.

Improved access to basic operating system capabilities eliminates the need to maintain and distribute a distinct C++ component as part of the Briefing Associate.

The .Net runtime environment provides means for *dynamic* creation of new classes. In theory (although we have not yet attempted it) this means that the Briefing Associate will be able to deal with descriptions of individuals belonging to multiple unrelated classes, by dynamically creating a class that implements the interfaces of each of them.

Considerable reworking of the mapping from ontology terms to OO classes and methods was done in this update to provide a more strongly-typed OO model and to simplify the code needed to implement model analyzers.

3. SEMANTICWORD: WORD BASED BRIEFING ASSOCIATE

SemanticWord offers an environment for authoring annotated text documents based in MS Word. Its aim is to reduce the burden involved in authoring semantic annotations. Authors are given a familiar and uniform environment where the creation of content and semantic descriptions can be freely interleaved. In many case both of them can be achieved in a single operation.

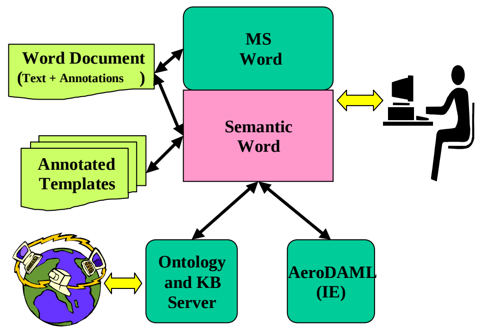


Figure 6. SemanticWord Architecture

3.1 Overview

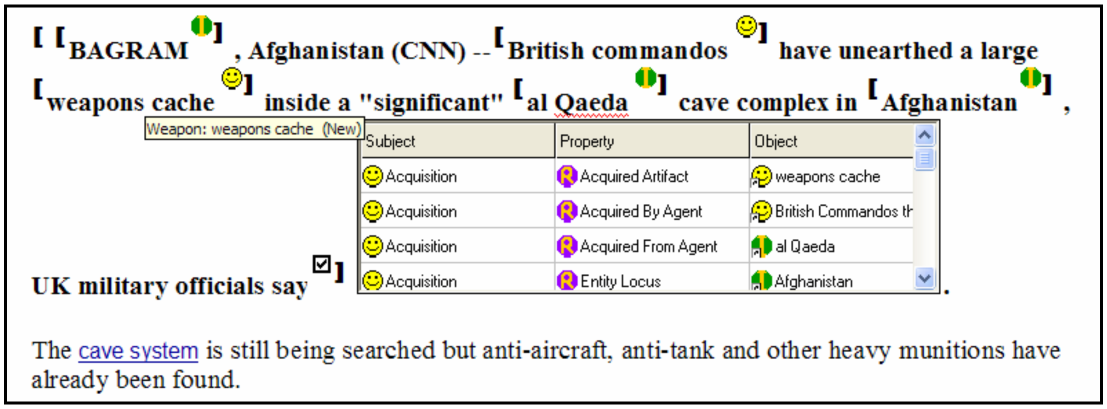
SemanticWord extends MS Word in several dimensions (see Figure 6). First, MS Word GUI is augmented with toolbars that support the creation of semantic descriptions (or annotations) that are attached to text regions. The GUI is also extended to show these annotations embedded within the text and to support their direct manipulation through mouse gestures. Second, SemanticWord extends Word's reach by opening a channel to the Semantic Web. Content from the Semantic Web (both ontology definitions and factual descriptions) is brought into SemanticWord to compose annotations that are later dumped back into the Semantic Web. Third, SemanticWord extends Word services by integrating AeroDAML, an automated information extraction system. AeroDAML analyzes and annotates the text of the document as it is being typed, appearing to the author as a service analogous to Word's spelling and grammar checking. Finally, SemanticWord supports the rapid composition of annotated text through template instantiation.

The above extensions were implemented using standard Microsoft extensibility technology. Annotations are rendered with ActiveX controls that can be placed in a document, implement their own behavior, control their GUI, and save their internal state. Automatic text analysis is driven by SmartTags technology that supports background parsing and tagging of the document text as it is being typed. The rest is supported by an Office COM Add-in that responds to MS Office/MS Word built-in events (e.g., DocumentOpen) and extend Word's menus and toolbars. The document content is manipulated through Word's COM API.

3.2 Semantic Annotations

SemanticWord annotations are based in the OWL (or DAML+OIL) language. SemanticWord annotations are attached to regions of text, not to the document as a whole. There are two types of annotations: *instances references* and *triple bags*. An instances reference associates a text region with a "referenceable" instance of a class. Triple bags describe the content of a text region with a collection of triples that follow OWL's *subject-predicate-object* model. The subject is an instance, the predicate is a property defined in an ontology, and the object can either be an instance or a value. SemanticWord Annotations are retained across text copy/cut and paste operations.

Figure 7 illustrates a fragment of an annotated document. An instance reference is rendered by enclosing the annotated text between square brackets and with an icon adjacent to the closing bracket. A triple bag is rendered by enclosing the annotated text between square brackets and displaying a checkbox and a triples table adjacent to the closing bracket. The checkbox allows the user to display or hide the table. To facilitate the handling of heavily annotated documents, the text associated to an individual annotation can be highlighted and all annotation marks can be made invisible.


[[BAGRAM] , Afghanistan (CNN) -- [British commandos] have unearthed a large [weapons cache] inside a "significant" [al Qaeda] cave complex in [Afghanistan] ,

Weapon: weapons cache (New)

Subject	Property	Object
Acquisition	Acquired Artifact	weapons cache
Acquisition	Acquired By Agent	British Commandos th
Acquisition	Acquired From Agent	al Qaeda
Acquisition	Entity Locus	Afghanistan

UK military officials say] .

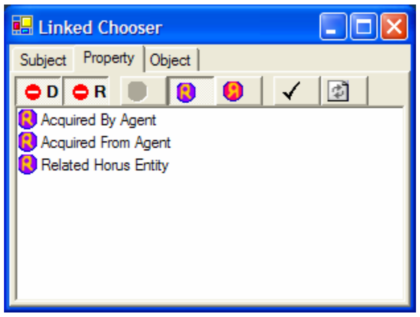
The cave system is still being searched but anti-aircraft, anti-tank and other heavy munitions have already been found.

Figure 7. Fragment of an annotated document.

The circular icon containing an I Bar (like that adjacent to "BAGRAM") references an external instance from the semantic web. A smiley face icon (like that adjacent to "weapons cache") references a locally defined instance. The boxed legend below the "weapons cache" instance reference is its tool tip. If the instance icon in the subject or object column of a table is overlapped by a small arrow in its lower left corner (like the one in the object column of the first row) then the cell is linked to an instance reference annotation. Modifying or deleting the linked instance reference will affect the triple too. If the instance icon is not overlapped by a small arrow (like the one the subject column of the first row) then the cell contains a direct reference to an instance.

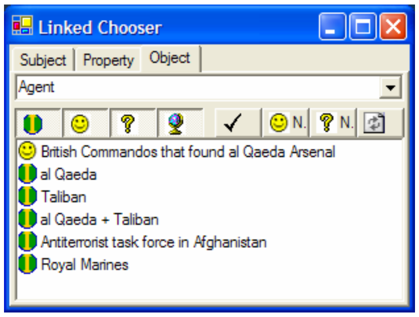
An Instance reference icon can be dragged and dropped over a cell corresponding to the subject or object of a triple. Cells filled using this method do not store a direct reference to the dropped instance but rather establish a link with the dragged instance reference. Updating the linked instance reference to refer to a different instance will alter the triple too. This level of indirection improves maintainability.

Triple cells can also be filled by picking instances and properties from special purpose browsers called *choosers* (See Figure 8). Choosers can use the values already stored in a triple to constrain the lists of choices offered to the user. For example, if the subject and object of a row are already filled in then the corresponding property chooser will only show the properties whose domain and range are consistent with those entries. Because SemanticWord does not enforce consistency, these constraints can be relaxed. The choosers also provide other filters for constraining the choices shown. For example, the instance chooser includes filters for listing only the instances that have already been referenced in the document. The instance choosers can selectively list instances corresponding to preexisting semantic web markup (provided by the Ontology and KB Server) or new instances defined in the current document. They also allow users to *create* new locally defined instances or provisional instances (described below), a function that a user would invoke if the listed choices do not include the desired instance. SemanticWord does not impose any order for filling in table cells, and can persist the state of tables containing rows with one or more empty cells.



Acquired By Agent
Acquired From Agent
Related Horus Entity

Property Chooser



British Commandos that found al Qaeda Arsenal
al Qaeda
Taliban
al Qaeda + Taliban
Antiterrorist task force in Afghanistan
Royal Marines

Instance Chooser (Object)

Figure 8. Property and Instance Choosers.

The choices correspond to the filling of the property and object columns of the second row of the triples table of Figure 7. The listed choices are constrained by the content of the other cells of the selected triple. These filters can be relaxed by toggling the buttons on the top toolbars. The Instance chooser also supports the definition of new instances.

Locally defined instances are instances that cannot be referenced from outside the document. Provisional instances are an artifact to postpone the identification of an instance that is being used to describe relationships. Ultimately, provisional instances must be replaced by references to external or locally defined instances. SemanticWord keeps track of the provisional instances and assists users in replacing them.

One obstacle that we noticed in other systems when composing a triple is that the role of the instances in the triple cannot be established before examining the definition of the predicate property. For example, determining who is the subject and who is the object in the relationship between an employee and her employer depends on how the property that relates both of them is (arbitrarily) defined. Assigning an instance to the subject or the object of a triple prematurely might preclude the possibility of establishing the relationship. To avoid this problem in SemanticWord, the property chooser can optionally list *reversed properties*. Reversed properties are ordinary properties that assume the subject and the object of a triple are switched. Reversed properties is only an artifact to add another degree of liberty in the order in which the triple arguments are filled -- the generated OWL markup switches the subject and object of a triple when a reversed property was selected.

3.3 Taming Annotation Authoring

SemanticWord was conceived with the goal of minimizing the burden involved in authoring semantic annotations. This burden is reduced through several techniques.

3.3.1 Non Intrusive Annotation Environment

SemanticWord provides an environment for authoring semantic annotations that is tightly integrated to MS Word. Word is the most massively adopted product for authoring text documents. SemanticWord includes a set of tools that economize the production of semantic descriptions and exploit opportunities for the simultaneous generation of text and annotations. Two examples of these tools are *personal class toolbars* and the *cascading class menus,* both illustrated in Figure 9.

Personal Class Toolbars: Personal Class Toolbars constitutes a convenient tool for generating both content and annotations together with just one mouse click. Users can create any number of Personal Class Toolbars, each one of them tied to a single class. Each personalized class toolbar contains an *instance selection* combo box and buttons to create instance references corresponding to the selected instance or a new one. If at the time the user creates an instance reference the document contains a selected region of text, then the instance reference will be attached to that region. If no text is currently selected, then both the "label" of the instance reference will be inserted in the document at the current text insertion point, and the new reference will be associated with the inserted text.

Personal class toolbars save effort when a small percentage of classes or instances account for a substantially larger percentage of the instance references that an author will need.

Classes Cascading Menu: A cascading class menu includes an entry for every named class in the ontology attached to the document. This menu gives users access to most of the operations related to ontology classes, including defining new instances, creating personal class toolbars, and opening instance choosers. When a user executes any of these functions from this menu, the menu entry corresponding to the selected class is duplicated and placed at the top of the menu so the user can access it easily the next time that she needs it. The cascading hierarchy is determined by the subclass hierarchy of the ontology. Classes with multiple superclasses appear in the cascade under each superclass.

Direct Manipulation of Annotations: Direct manipulation of annotations is another method of simplifying the production of semantic annotations. In SemanticWord users can compose semantic annotations by manipulating other annotations that are placed within the document. For example, the subject and object of a triple can be filled by dragging instance references annotations over the triple. For some users this method is faster and more natural than searching for those same instances in instance browsers.

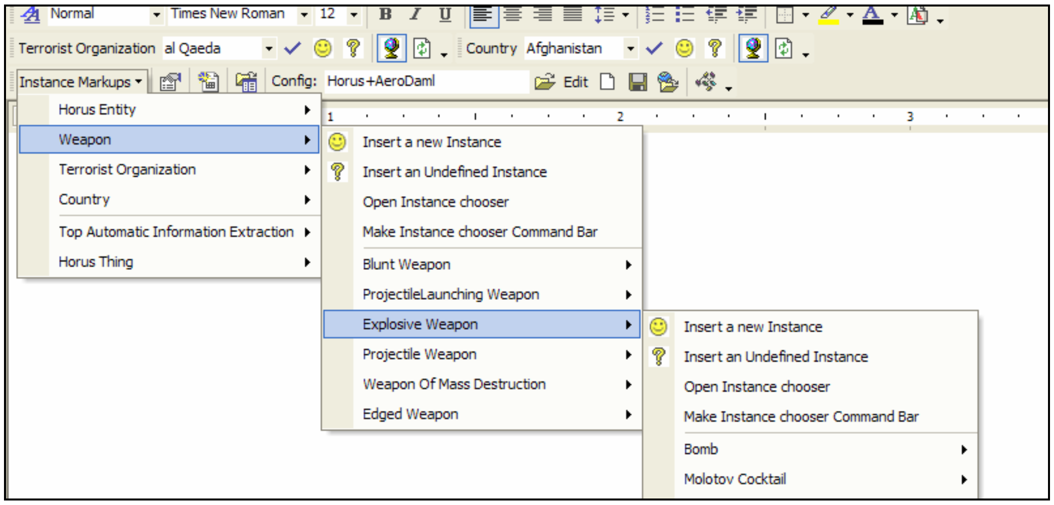


Figure 9. Toolbars and Menus.

The last two toolbar rows belong to SemanticWord. The first row contains two juxtaposed *personal class toolbars*. The first one is tied to the class "Terrorist Organization" and has selected the instance "al Qaeda". The second one is tied to "Country" and has selected "Afghanistan". Clicking in the Check button will generate both the text and the annotation corresponding to the selected instance. The other buttons are for defining new instances before inserting their text and annotation. The last toolbar row has its *classes cascading menu* opened. This menu provides access to several class related functions. The most recently chosen classes get added to the top of the menu (like Weapon, Terrorist Organization, and Country) for easy access.

3.3.2 Flexible commitment order

Authors should not be forced to follow a strict order in carrying out the many steps involved in authoring semantic descriptions. Many of the features that support this principle have been introduced before. These features are summarized in this section.

Elements of a triple can be entered in any order. Even the determination of which instance is the subject and which is the object can be postponed by means of the **reversed properties**. New instances can be created from the instance choosers avoiding a disruption of the triple's composition process. Unlike other annotation tools, triples are laid out in a tabular structure rather than in a tree or other structures that impose a topological dependency among its nodes.

Consistency is not enforced. A user is free to compose a triple that violates ontology constraints. The user can make the changes that would fix this conflict at a time convenient to her. Consistency is taken into account when filtering suggested choices for composing a triple, but the user can deactivate these filters with a single button click.

Instance identification can be postponed but the instance can still be used to describe relationships. This is achieved through the use of **provisionary instances**, which can be used wherever definitive instance can but remind the user of the unconcluded task. SemanticWord will assist users in assigning identity to these instances.

3.3.3 Annotation Reuse

Annotations are attached to text regions and are going to be reused when those regions are reused. In particular, annotations are carried over along text cut/copy and paste operations and when fragments of a document are reused elsewhere in the same document in other documents based on the same ontology.

3.3.4 Automatic Information Extraction

SemanticWord integrates an Information Extraction System (IES). Automatic information extraction technology promises to significantly reduce the human overhead involved in the semantic annotation task. Although this technology has not reached a level of sophistication required to capture deep relationships in text, it can provide useful annotation fragments. The approach taken in SemanticWord is to supply the tools that would allow users to augment the annotation provided by an IES.

SemanticWord uses AeroDAML, an IES developed at Lockheed Martin. AeroDAML processes text and produces OWL markup that relates instances and values to Ontology classes and types. AeroDAML relies on a high performance commercial information extraction system called AeroText. The default AeroDAML is based in the default AeroText which includes "domain independent" extraction rules capable of extracting many proper nouns and frequently occurring relations. AeroText and consequently AeroDAML can be tailored to particular domains through training sessions with annotated corpuses.

SemanticWord provides an environment for refining and augment the result of IESs. We observed that the default AeroDAML does a good job at recognizing and categorizing proper nouns but their classification tends to be overly general. It also fails to recognize most of the relations between instances. For example, AeroDAML succeeds in classifying **Kabul** as a **Place** but failed in finding the more specific class **City**, perhaps because there was nothing in the text that might clue AeroDAML about this fact. SemanticWord let AeroDAML to recognize and classify proper nouns but expects the user to refine the classification and to specify their relationships.

SemanticWord drives the information extraction process on the fly. As the user types the content of the document, a background thread feeds new or modified text to AeroDAML in paragraph units (roughly), obtains the extracted entities with their position in the text, and underlines those text regions with a blue wiggly line. This procedure is carried out in a way that resembles Word spelling and grammar checking and is implemented in terms of Microsoft SmartTags technology.

The user can examine the extracted entities and convert them into instance reference annotations. As part of this conversion the user has the option of refining the extracted type. Once an extracted entity has been transformed into an instance reference it behaves just like a natively created instance reference. In particular, it can be dragged and dropped onto cells of triple bags to describe the relationships that AreoDAML missed.

Our original interface to AeroDaml was noticeably unresponsive due to the web-based implementation of the AeroDaml server, and its requirement that text be submitted to it indirectly, via the URL of a document on some other web server. We obtained a trial license of the AeroDaml *client/server* implementation, which can run on the same machine as MS Word. This implementation's protocol allows text to be submitted directly to the server.

The web-based AeroDaml did not include in its markup an explicit correlation between the markup and a range of text to which the markup applied. Heuristics were required to assign the markup to any text range smaller than the *entire* submitted text. We worked with the AeroDaml developers to define a protocol, which they implemented, that included in the XML returned by AeroDaml an explicit correlation.

AeroDaml also infers coreference across non-contiguous text ranges. We have recognized that the basis for these inferences can easily be lost as text is edited or in cut/paste operations, but are still considering how to deal with this issue.

3.3.5 Annotated Templates

Annotated text templates reduce the amount of work involved in authoring both semantic annotations and document content. A template consists of a text fragment annotated with semantic and template related descriptions, and persists as a (typically quite small) word document.

A template may be inserted into a document just like any other document. Both the text and annotations of the template are copied into the target document. After insertion, the copy can still be subjected to further editing and annotating.

Templates are authored in SemanticWord in template design mode. All annotations tools described previously are also available for annotating templates in template design mode plus an additional toolbar that includes the template specific authoring tools described below. We expect that non-programmers would be able to author templates.

Instance Placeholder: An instance placeholder annotates a region of text that needs be replaced by an instance reference when the template is used in a document. It also serves as the surrogate for an instance reference, and as such, it can participate as the subject or the object of one or more triples in the template's triple bags.

An instance placeholder is rendered like an instance reference annotation but with a different icon. In design mode this icon can be dragged over triple tables to compose the semantic annotations that describe the template. It can also be dragged over another instance placeholder to specify a co-reference requirement. In instantiation mode, this icon is a drop site for the concrete instance that is going to be bound to the instance placeholder.

When an instance placeholder is bound to an instance reference, the label of the instance reference replaces the template's text and all co-referential instance placeholders are bound to that instance.

Optional group: An optional group delimits a region of text in the template that can be optionally included in the instantiation of the template. The text delimited by an optional group can contain annotations and other groups. In particular, it can contain instance placeholders. Opting to delete an optional group from an instantiated template will automatically remove any triples having a cell linked to an instance placeholders within the deleted group.

Repeated group: Like an optional group, repeated group annotation delimits a region of text and can also contain other groups and annotations. During instantiation the user can ask that a repeated group be replicated any number of times. Each replication of the group creates its own incarnation of the instance placeholders that it contains. When the group is replicated, all triples with cells linked to the instance placeholders contained in the group are replicated as well.

The utility of annotated templates is enhanced by the IES described above. The IES analyses the document and generates instance reference annotations corresponding to the concrete entities mentioned in the text. These instance references can be dragged over the template instance place holders to instantiate the template and generate instantiated triples describing their relationships.

3.4 Annotating TEXT Regions

In SemanticWord, semantic descriptions are distributed throughout the document and attached to text regions that "support" their content. This is not a requirement for the semantic web. Most of the semantic markup authoring tools reported in the literature do not adopt this practice. The descriptions they produce are associated only with a document, not with portions of that document.

We speculate that relating a semantic description to the text that supports it has advantages in terms of annotation authoring, reuse, maintenance, and validation. However, we also recognize that this practice might introduce unnecessary complications.

Some of the advantages of attaching semantic descriptions to text are:

Descriptions can be reused if the text is reused. Annotations are carried over along text cut/copy and paste operations and when document fragments are reused in other documents.

Conformity between the semantic descriptions and the content of the document can more easily be validated and maintained.

Authors might find it natural to find annotations by finding, through familiar text search/scroll mechanisms, the text to which the annotations are attached. Contrast this with browsing the semantic markup directly. For example, in SemanticWord authors compose triples by dragging around instance references placed within the text.

Markup that is tied to text fragments disappears if the text fragment, or a region containing it, is deleted. Generally, this is desirable because the document's content no longer supports the statement formalized by the deleted annotation.

Among the difficulties of this approach we found:

If an entity (e.g., a person or place) is mentioned several times within the text, it might be necessary to duplicate its annotation too.

Some concepts might be implicit or too abstract to be located in the text.

As changes are made to text within an annotated region – particularly at its boundaries – heuristics must be used to adjust the boundaries. The use of paired brackets for rendering these regions keeps the user informed of the result of these heuristics.

Although SemanticWord is biased toward the attachment semantic annotations to text, it does not mandate it, opening a whole spectrum of hybrid compromises. For example, authors might choose to attach instance references to text but to describe their relationships in a single global triple bag. Moreover, not even the instance reference annotations are required because the triples can be filled directly from instance choosers. More serious use of SemanticWord will be required to weigh the pros and cons of this approach.

3.5 Controlling Scaling

We identified *scaling* as a major research issue. We see three dimensions in which issues of scale arise: the size of the document being annotated, the size of the ontology being used as the terminology for the markup, and the size of the knowledge base containing existing facts encoded using that markup. Our solutions are driven by the requirements of the task outlined above.

The summaries to be produced are relatively short – one or two paragraphs – and the documents to be summarized are themselves relatively short – several pages. We anticipate that a few simple, generic GUI features now being implemented can easily deal with this scale. The primary solution is presenting the summary being composed in a window/pane separate from the document itself, to ensure that a drag site and its target drop site can be simultaneously visible. A few simple, ontology-guided, navigation tools will suffice to let authors quickly find likely candidates for template variable instantiation in documents of this size.

The HORUS ontology and the HORUS knowledge base are both sufficiently large that we do not see generic solutions that will be satisfactory. Our approach is to provide a *configurable* environment. We will capture, via annotations to the HORUS ontology, some notion of the *area of specialization* of an intelligence analyst. This area of specialization will give us a reduced ontology (subset of classes and properties) and a reduced knowledge base (subset of individuals and assertions) within which scaling is not a problem. The analyst will be able to "escape" from his subarea to the full ontology and/or knowledge base, but when he does so he will be forced to deal with larger menus and deeply nested concepts.

This work relies on two hypotheses that need validation:

that *reusable* templates can be built for the kinds of documents that (many) intelligence analysts produce

that sufficiently small areas of specialization can be defined for individual analysts such that they confine (almost) all their markup to that subarea

Although these hypotheses are in accord with the intuitions of our HORUS colleagues, we cannot validate them ourselves. For example, although we can build sample templates using our template construction tools, we cannot realistically determine what could be stated in a *reusable* template, because we are not intelligence analysts and do not even have access to the classified documents they produce. Validation of the hypotheses will require the active participation of the HORUS technology staff, who do interact with analysts.

We have created suitable abstract ontology and knowledge base interfaces. Both have been implemented for the Jena.Net (BBN) substrate. The KB interface was specified in such a way that it can easily be implemented in terms of DQL when DQL servers become available.

4. SEMANTIC MARKUP IN INTERNET EXPLORER

During the project we recognized that the OWL markup capabilities we were developing for MS PowerPoint and Word could also be incorporated into other COTS products – in particular Internet Explorer – and that by doing so we would be filling a huge void for a COTS-based semantic web markup tool (as all existing semantic web markup tools are research prototypes).

4.1 A Metaphor for Semantic Markup

To realize the vision of the Semantic Web we must provide tools that enable people to semantically markup the information being published on the web in a simple and efficient manner. Such tools currently don't exist for web pages containing unstructured data. Instead, tools are being developed for translating the structured machine-readable data in databases or reverse engineering the semi-structured data available on the web into semantic markup. But semantic markup for unstructured web pages remains a largely unexplored and unsupported area.

Semantic markup is neither human readable (by non-computer gurus) nor writeable. We must therefore use tools to mediate our interaction with semantic markup to make it easy to read and write. We have chosen a "triples" metaphor for interacting with semantic markup and a table interface for viewing and manipulating these semantic markup triples. This semantic markup can be attached to any contiguous region of text in the page being marked up and remains attached to that region as the page is published and/or modified.

This metaphor and interface hide many of the complexities of semantic markup by using mnemonic names for URIs, hiding unnamed intermediate objects (represented by "GenSym" identifiers), and allowing multi-valued attributes and embedded objects. It closely approximates the triples metaphor used within semantic databases while avoiding the syntactic restrictions imposed on the external, or published, representations of these triples stored in files or passed between tools in the OWL (or DAML+OIL) standards.

When a semantically marked-up page is saved, these markup triples are converted into the OWL external tag-based format and embedded within the page's Hyper Text Markup Language (HTML) so that it is anchored to the text and/or location in the page that it is annotating. Any OWL compliant tool that subsequently accesses the page can then extract them, and the annotated text.

4.2 Annotating Ontologies

When viewing or creating semantic markup, we expect users to be operating in the context of ontologies created by others. It is therefore inevitable that they will find certain terms defined by those ontologies awkward, hard to recognize, or ambiguous and prefer to use their own "names" or "nicknames" for those terms.

But users can't change someone else's ontology[1]. Instead, they need a simple way to customize it for their own use. Annotations – local statements about defined concepts – provide such a facility. They allow a user to customize an ontology (without modifying the original) as long as all the tools in which that customization is desired know where to find those annotations and how to interpret them.

But that's the problem with annotations. Standards are just beginning to emerge for where to place them (or alternatively how to find them) or how to interpret them. We have nevertheless chosen them as the means for customizing ontologies for semantic markup as both the location and interpretation problems can be trivially solved for a single tool – our browser Plug-In – by having it place them where it can later find them and store them in a form it can later use for interpreting them.

We've also attempted to simplify both problems by storing them as OWL markup (so they can be searched for) and express them as overrides or additions to the original ontology (so they are directly interpretable in terms of that original ontology).

Ideally one would like to use ones own annotations as well as those developed by others in some preferential order (e.g. mine first, then those of my department, then those of my organization, then those of the ontology), but we haven't addressed this issue.

We've identified three types of ontology customization that facilitate the understanding and creation of semantic markup – names, icons, and structured displays – discussed below. Only the first of these is addressed by current ontology standards, but the other two could easily become standards once their use became apparent.

4.2.1 Customizable Names

Users can choose to override the name given to a type or property in an ontology, or the name of an instance defined by that ontology, with a local name of their own choosing. Presumably that local name will be more mnemonic, shorter, and/or less ambiguous – at least for the user – than the original.

Label is the standard OWL concept used for specifying the human consumable name for a concept, and when available it is used in the triples metaphor as the displayed "name" for a concept. Otherwise the resource's local name is used. Either of these names is overridden by any customized name chosen by the user.

4.2.2 Customizable Icons

As described in the next section, much of the activity of creating markup involves selecting the appropriate type, property, or instance from some list or set of buttons. To facilitate those choices the lists must somehow be pruned to a manageable size (on the order

[1] They could of course define a new ontology derived from the original that makes the desired substitutions. But that is far too much work and hides the fact that they are using the original ontology.

of twenty or fewer elements) from which the user can select (see the Pruning Markup Selection Lists section for a discussion of techniques for keeping the size of these lists manageable).

However, even when the size of these lists is kept manageable, it still takes people a significant amount of time and effort to find an already mentally selected item within a textual list (because they have to read each element in the list or find the appropriate place in an alphabetically sorted list). When familiar and recognizable graphic icons are used with the text labels, finding the desired element is much faster and easier, particularly when those icons were chosen by the user.

Users are therefore allowed to associate icons with OWL types, properties, and instances. Once that association is formed the chosen OWL concept is always displayed with its associated icon (on toolbar buttons the icon is displayed and the associated OWL concept appears as a "tooltip" when the mouse hovers over the button). These associations are also stored persistently so that they can be used in subsequent markup sessions.

Though it has not yet been done, one can imagine the standardization of how such icon associations could be directly incorporated into an ontology by its creator so that they would be available to all users of that ontology as a graphic aid in recognizing the types and properties defined by that ontology.

4.2.3 Customizable Structured Displays

While the triples metaphor is quite general, understanding the meaning of a triple from its three parts is often difficult.

However, when those three concepts are embedded in a natural language phrase, the meaning becomes obvious. As an example, consider the following example triple taken from Mike Dean's Homepage (we've expressed it as a triple rather than use its original OWL syntax, and used short names instead of URIs):

<DeanBS, school, Stanford>

which is intended to mean that Mike Dean's Bachelors Degree was granted by Stanford University.

Now consider the following annotation:

<Degree, school, > = "%1 was granted by %3"

This annotation states that for a triple whose subject is of type "Degree" and whose property is "school" (the blank after the second comma indicates no restriction on the object slot of the triple) the triple should be displayed using the natural language phrase on the right of the equals sign with "%1" replaced by the subject of the triple, "%2" replaced by the property of the triple, and "%3" replaced by the object of the triple.

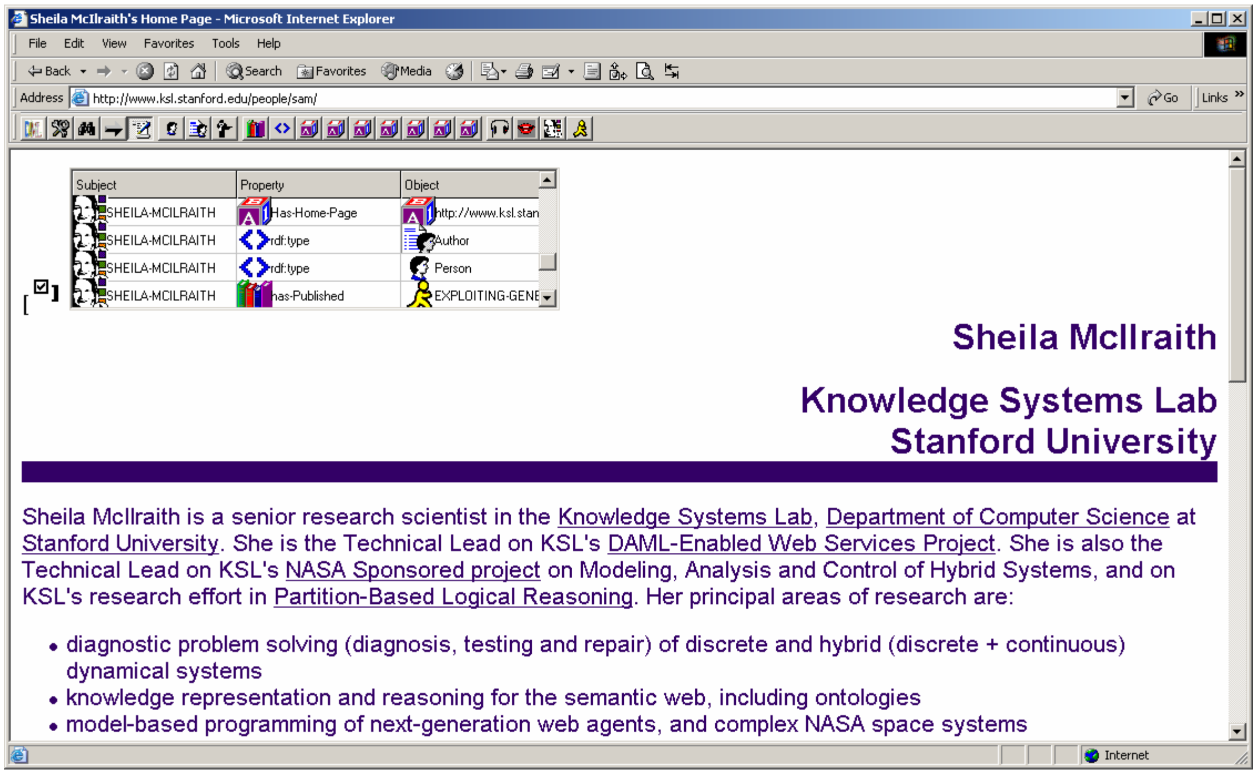
For the example triple this would yield "DeanBS was granted by Stanford." This structured display is shown as a "tooltip" when the mouse hovers over the property of a triple for which there is an applicable structured display annotation.

4.3 A Browser PlugIn

Rather than developing a standalone tool for viewing, creating, and modifying semantic markup, we felt it was important to provide this capability from within the standard tool for viewing web pages – the web browser. We chose Microsoft's Internet Explorer to be the host for our tool because it is the most widely used web browser and because it had an extensibility architecture that allowed our tool to be developed as a Plug-In.

As a Plug-In our tool operates within the browser and has access to the page being displayed and various events that occur such as a new page being loaded. The Plug-In is only concerned with semantic markup and is inactive on pages that don't contain semantic markup. The browser performs all loading and rendering of pages.

4.3.1 Viewing Existing Semantic Markup


Sheila McIlraith's Home Page - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://www.ksl.stanford.edu/people/sam/
Go Links
Internet

Subject	Property	Object
SHEILA-MCILRAITH	Has-Home-Page	http://www.ksl.stan
SHEILA-MCILRAITH	rdf:type	Author
SHEILA-MCILRAITH	rdf:type	Person
SHEILA-MCILRAITH	has-Published	EXPLOITING-GENE

Sheila McIlraith

Knowledge Systems Lab
Stanford University

Sheila McIlraith is a senior research scientist in the Knowledge Systems Lab, Department of Computer Science at Stanford University. She is the Technical Lead on KSL's DAML-Enabled Web Services Project. She is also the Technical Lead on KSL's NASA Sponsored project on Modeling, Analysis and Control of Hybrid Systems, and on KSL's research effort in Partition-Based Logical Reasoning. Her principal areas of research are:

- diagnostic problem solving (diagnosis, testing and repair) of discrete and hybrid (discrete + continuous) dynamical systems
- knowledge representation and reasoning for the semantic web, including ontologies
- model-based programming of next-generation web agents, and complex NASA space systems

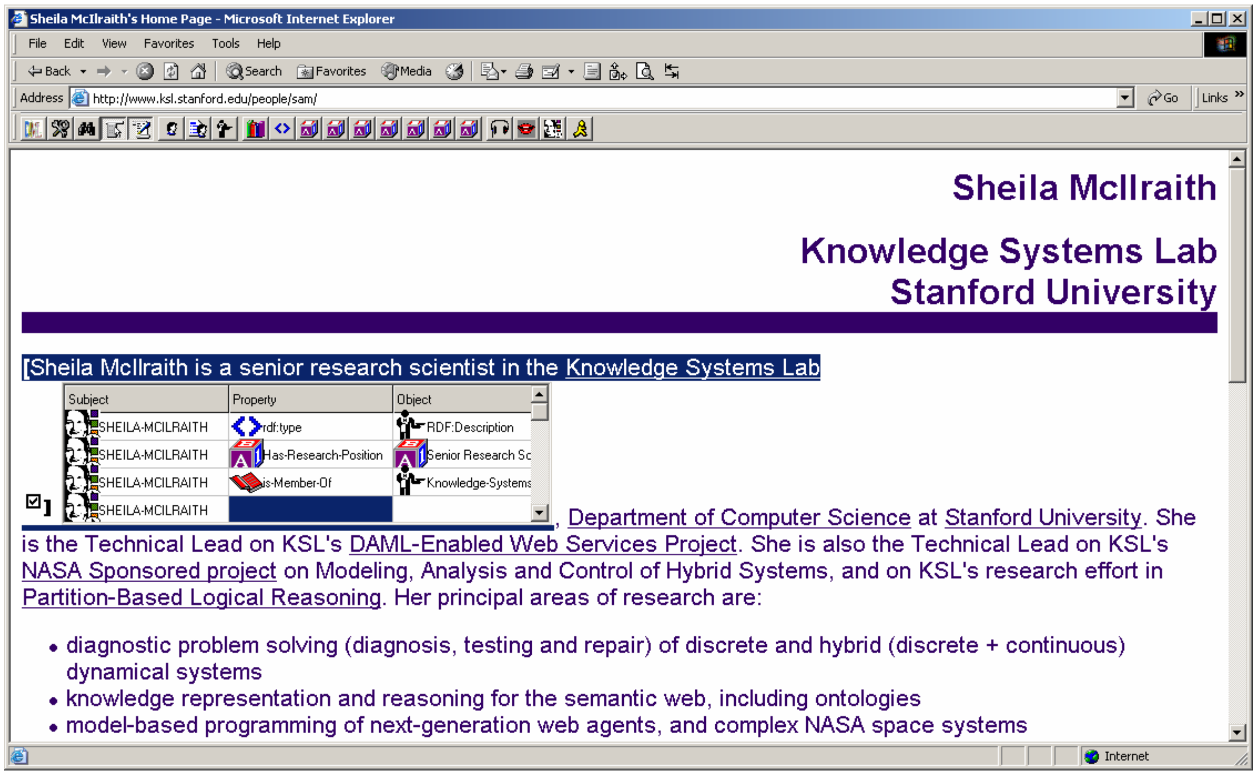
Figure 10. Viewing Existing Semantic Markup

When a page is loaded into the browser, our Plug-In scans the page (through the browser's internal document object model) to see if it contains any existing semantic markup. If so, the Plug-In identifies the OWL types, properties, and instances used in that semantic markup and displays a toolbar containing buttons corresponding to these OWL concepts used in the existing semantic markup.

These concept buttons serve as single-click search commands to find the next use of the selected concept in the semantic markup scanning forward (Search-Forward mode) or backward (Search-Backward mode) from the current location in the page. The found semantic markup is displayed in place in the page as a Semantic Markup Table that shows either the single triple containing the selected concept (Concise mode) or all the triples contained in the found top-level semantic markup block (Complete mode). The text that the semantic markup annotates is highlighted and immediately precedes the Semantic Markup Table.

4.3.2 Creating New Semantic Markup

Semantic markup can be added to a page by simply selecting the contiguous range of text to which the semantic markup will be attached and selecting the first OWL concept to enter into the new semantic markup. A new top-level semantic markup block is created in the page, attached to the selected text, and initialized with the selected OWL concept. A Semantic Markup Table is displayed in place in the page with this initial OWL concept. The rest of the initial triple and as many additional triples as desired are filled in by selecting the desired OWL concepts for each cell of the Semantic Markup Table. As each cell is filled in, the corresponding semantic markup is created within the page.


Sheila McIlraith's Home Page - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://www.ksl.stanford.edu/people/sam/
Go Links
Internet

Sheila McIlraith

Knowledge Systems Lab
Stanford University

[Sheila McIlraith is a senior research scientist in the Knowledge Systems Lab

Subject	Property	Object
SHEILA-MCILRAITH	rdf:type	RDF:Description
SHEILA-MCILRAITH	Has-Research-Position	Senior Research Sc
SHEILA-MCILRAITH	is-Member-Of	Knowledge-Systems
SHEILA-MCILRAITH		

], Department of Computer Science at Stanford University. She is the Technical Lead on KSL's DAML-Enabled Web Services Project. She is also the Technical Lead on KSL's NASA Sponsored project on Modeling, Analysis and Control of Hybrid Systems, and on KSL's research effort in Partition-Based Logical Reasoning. Her principal areas of research are:

- diagnostic problem solving (diagnosis, testing and repair) of discrete and hybrid (discrete + continuous) dynamical systems
- knowledge representation and reasoning for the semantic web, including ontologies
- model-based programming of next-generation web agents, and complex NASA space systems

Figure 11. Viewing New Semantic Markup

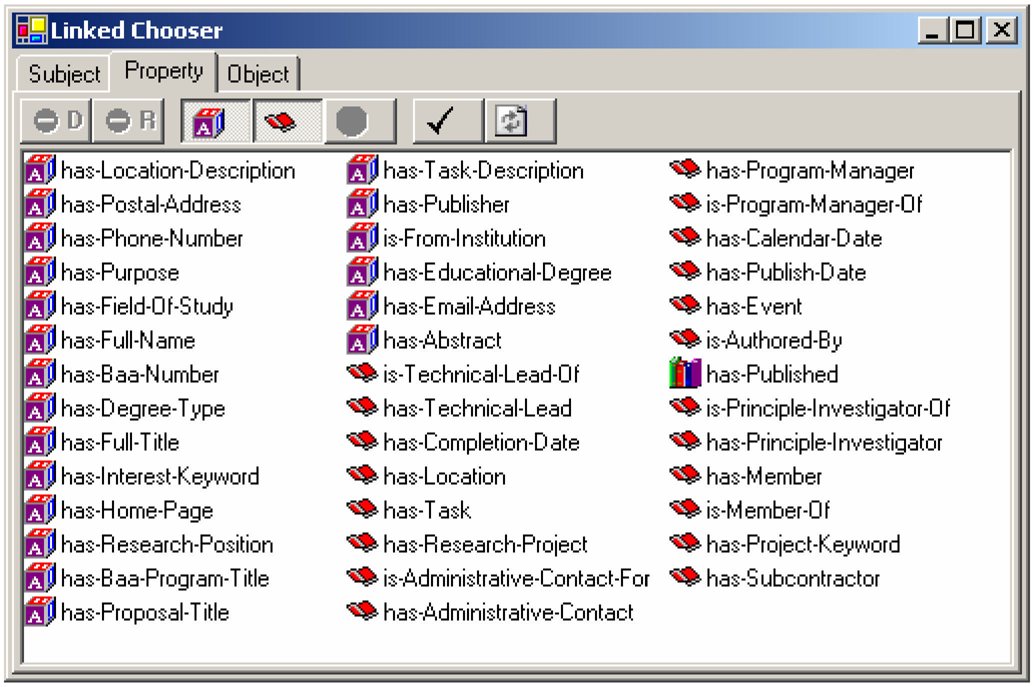
Figure 11 displays the semantic markup created to describe the highlighted text. The first row is an *RDF:Description about* the person Sheila McIlraith. It was created when the icon for Sheila McIlraith was pushed (second button from the right in the OWL toolbar) after selecting the text to be marked-up. This *RDF:Description* acts as a wrapper for the rest of the markup being created. The second row indicates that her research position is senior research scientist while the third row indicates that she is a member of the Knowledge

Systems Lab. The OWL output when the page is saved corresponding to this semantic markup is shown in Figure 12.

```
<rdf:description has-Research-Position="Senior Research Scientist"
                 rdf:ID="ksl-daml-desc.daml#SHEILA-MCILRAITH">
     <FONT face="Arial, Helvetica, 'Sans Serif'">
          <FONT color=#330066>
               Sheila McIlraith is a senior research scientist in the
               <A href="http://www.ksl.stanford.edu/">
                    Knowledge Systems Lab
               </A>
          </FONT>
     </FONT>
     <is-Member-Of rdf:resource="http://ksl.stanford.edu/projects/DAML/
                       ksl-daml-instances.daml#Knowledge-Systems-Lab"/>
</rdf:description>
```

Figure 12. Generated Semantic Markup

Notice that it surrounds the text it describes. This annotated text was automatically segmented from the larger text range contained in the original page (as shown in Figure 10) when the markup was created and retains all the HTML tags contained by the original (here the two *FONT* tags and the embedded *Anchor* reference to the Knowledge Systems Lab) as does the remaining portion of the original text range. Also notice that the two triples defined in this semantic markup are represented quite differently within the generated output – *has-research-position* is defined as a datatype property by the ontology used by the page's author and appears as an attribute within the *RDF:Description* tag while *is-Member-Of* is defined as a object relation and appears as a separate tag within the *RDF:Description* block.



has-Location-Description	has-Task-Description	has-Program-Manager
has-Postal-Address	has-Publisher	is-Program-Manager-Of
has-Phone-Number	is-From-Institution	has-Calendar-Date
has-Purpose	has-Educational-Degree	has-Publish-Date
has-Field-Of-Study	has-Email-Address	has-Event
has-Full-Name	has-Abstract	is-Authored-By
has-Baa-Number	is-Technical-Lead-Of	has-Published
has-Degree-Type	has-Technical-Lead	is-Principle-Investigator-Of
has-Full-Title	has-Completion-Date	has-Principle-Investigator
has-Interest-Keyword	has-Location	has-Member
has-Home-Page	has-Task	is-Member-Of
has-Research-Position	has-Research-Project	has-Project-Keyword
has-Baa-Program-Title	is-Administrative-Contact-For	has-Subcontractor
has-Proposal-Title	has-Administrative-Contact	

Figure 13. Markup Chooser for Triple Property

To simplify the selection of appropriate OWL concepts for the triples within a Semantic Markup Table, a Markup Chooser form is displayed that contains listings of the possible values that could respectively fill in the subject, property, and object slots within a triple. These values are obtained from the ontology used by the page and from any knowledge base (containing instances of the ontology defined types) referenced by the page. Figure 13 shows the Markup Chooser when the property column is selected in a new row of the markup table (i.e. the property is not yet constrained by the subject or object of that row). The possible datatype (or attribute) properties (denoted by the 3D-block icon) and object relation properties (denoted by the open-book icon) are listed and can be separately included/excluded by the *datatype* and *object relation* toolbar buttons (denoted by those same icons).

4.3.2.1 Pruning Markup Selection Lists

This potentially large space of types, properties, and instances is pruned by any constraints contained in the ontology. Moreover, these constraints are cumulatively applied as slots in a triple are filled in[2]. Thus, once the subject is filled in, then only properties whose domain restrictions are consistent with that choice are listed as possible choices for the property slot of that triple. If the object slot was also filled in then both domain and range restrictions would both be used to prune the set of possible properties for that triple.

[2] These automatic constraint prunings can be deactivated through controls on the Markup Chooser form.

Similarly, once the property slot is filled in, then the subject and object slots of that triple can be constrained respectively by the domain and range restrictions of the chosen property.

In addition to using these ontology constraints to prune the set of possible instances, the user can navigate through the type hierarchy defined by the ontology and list the instances of the selected type.

The user can also employ additional filters that limit the possible instances to those that are (already) used in semantic markup in this page, and/or those that were newly created in this session (i.e. execution of the Plug-In).

Alternatively, rather than selecting an existing instance, the user can of course create a new instance of any ontology defined type and use that new instance to fill the subject or object slot of a triple.

This Markup Chooser form is linked to the Semantic Markup Table so that the selected cell in the Semantic Markup Table determines whether the Markup Chooser lists potential subjects, properties, or objects and which ontology constraints can be applied to prune that list based on the other filled in values of that triple.

4.3.2.2 Automating Markup Entry

When creating semantic markup there are several regularities in how markup tables are filled in that can be used to automate the process so that users don't have to perform as many actions to fill in the markup table.

44.3.2.2.1 Automatic Cell Selection

The first is that cells are normally filled in left to right (i.e. subject followed by property followed by object) and top to bottom (i.e. triples are sequentially added one at a time). Thus, once a cell is filled in, if the next cell to the right (the subject of the next line for the object slot of a triple) is empty, then that cell can be automatically selected as the next to be filled in. As the Markup Chooser is linked to the selection made in the Semantic Markup Table, this automatic selection also positions it to the appropriate listing of possible subjects, properties, or objects.

Naturally, the user can manually select a different cell to alter the order in which the Semantic Markup Table is filled in.

4.3.2.2.2 Automatic Subject Filling

The second regularity is that the subject field of the Semantic Markup Table normally repeats the subject chosen in the first triple of the table. Recall that new Semantic Markup Tables are created by selecting an instance the markup is about and associating it with some piece of text in the page. That selected instance becomes the subject of the first triple in the Semantic Markup Table and subsequent triples usually provide additional information about that concept. Thus this cell can be automatically filled in as the subject cell is automatically selected once a previous triple is completed.

Naturally, this automatically entered value can be overridden if the user wants to create a triple about a different subject.

4.3.3 Plug-In Implementation

Our Plug-In is implemented as an Internet Explorer toolbar that is activated or deactivated from the Internet Explorer View/Toolbar menu (this choice is retained in future activations of Internet Explorer until changed again by the user).

In general, the browser's Plug-In architecture supported the integration of our Semantic Markup tool quite well. The browser retained control of all page loading and rendering of the page (other than the display of the cells of the Semantic Markup Table within the rectangular area reserved for it in the browser's rendering of the page), while the Plug-In provided the extra facilities required to view and create semantic markup within those browser pages.

However, there were three problems that arose from the browser's Plug-In architecture that are discussed below in this section after we first describe the successful incorporation of Semantic Markup Tables within the browser's rendering of a page.

4.3.3.1 Semantic Markup Table Browser Control

The Semantic Markup Table that displays the triples contained within a top level block of semantic markup is implemented as an ActiveX control that is dynamically inserted into the page to display that semantic markup. This Semantic Markup Table allows us to control how the semantic markup is displayed and how user actions (such as mouse-clicks and drag-and-drop) are interpreted to create and/or modify the semantic markup.

Moreover, by concentrating the semantic markup display and manipulations in the Semantic Markup Table, the page itself (including its markup) doesn't need to be altered to display markup (other than the insertion of the Semantic Markup Table as an ActiveX control). This simplifies the management of display markup as a single display can be moved from one markup block to another by removing it from the first block and reinserting it at the second markup block or additional displays can be added to other markup blocks while keeping the original in place. Saving a page with added or modified markup is also simplified by just excluding these Semantic Markup Tables from the export of the otherwise unaltered page.

4.3.3.2 PlugIn Implementation Problems

4.3.3.2.1 Upper Case Tags

By far the biggest problem encountered with our Plug-In implementation was that the browser does not preserve the case of tags it incorporated as it parses and internalizes web page markup. All of these tags are uppercased in accordance with the HTML specification (which unfortunately differs from the case-sensitive nature of XML, RDF, and OWL tags). Thus, the internal "document object model" that the browser makes available to examine and modify the page being displayed, does not contain the proper references to the ontologically defined OWL concepts (because those references have be upper-cased).

In particular BBN Technologies' Jena.Net, which is used to build an ontology knowledge base from which type hierarchies and domain and range restrictions are extracted, will fail to locate ontology concepts from references that have the wrong case. We must therefore create a mapping from the references that appear in the internal markup (the domain object model) to the case-correct references (URIs) that are used within Jena.Net.

This mapping is obtained by loading the page's ontology into Jena.Net and then (case-insenitively) matching the uppercased references against the appropriate set of ontology concepts (e.g. object-property references must match OWL properties and type references must match OWL types).

This problem does not occur with instance references since they don't occur as tags in the semantic markup. Thus, only the ontology needs to be loaded into Jena.Net to resolve the uppercased references.

4.3.3.2.2Non-Display of HTML Head

The Plug-In displays the Semantic Markup Table immediately after the text or place in the web page to which the semantic markup is attached. Unfortunately this is often within the header of the web page (as defined by the "HEAD" tag). In accordance with the HTML specification no displayable content is expected to occur within the HEAD tag and Internet Explorer gets an error when a Semantic Markup Table is placed there.

We therefore had to institute a check to determine whether the markup to be displayed occurred within the page header, and if so, to place the corresponding Semantic Markup Table at the beginning of the body of the page (so that it can be properly displayed).

4.3.3.2.3 Saving Semantic Markup

Although Internet Explorer has a *Save* operation, we had to implement our own for two reasons. First, because Internet Explorer is a *Browser* – rather than a web-authoring tool – it ignores the current state of the loaded document and instead saves a newly loaded version of the document. Second, Internet Explorer saves the document with uppercased tags rather than the tags originally contained in the document.

Our *Save* operation exports the current state of the document – including any semantic markup added to it – excluding any Semantic Markup Tables displaying that markup. Rather than unparsing the object model for the document itself, it utilizes the unparsed string representation of that object model maintained by Internet Explorer (in the *outerHTML* attribute of the document's top node) and scans that string to restore the case of uppercased tags and remove the presence of any Semantic Markup Tables.

4.3.4 Shared Infrastructure

The browser Plug-In makes heavy use of BBN Technologies' Jena.Net to store and access three separate semantic knowledge bases associated with the web page (see Figure 14). The first is the ontology knowledge base that is loaded from the ontology URI extracted from any semantic markup contained within the page. The second is the knowledge base (if any) of instances referenced by the semantic markup. It too is loaded from a URI extracted from the semantic markup. Finally, the existing semantic markup is extracted from the page and loaded into the semantic markup knowledge base. This markup must be extracted from the surrounding HTML because the current Jena.Net cannot parse and ignore the HTML while processes the embedded semantic markup.

These three knowledge bases are used to populate the set of toolbar buttons that correspond to concepts already used in the page's semantic markup (from the Semantic Markup KB), the set of types and properties listed in the Markup Chooser (from the Ontology KB), the set of instances listed in the Markup Chooser (from the Instance KB), and the subset of those instances already used in the page's semantic markup (from the Semantic Markup KB).

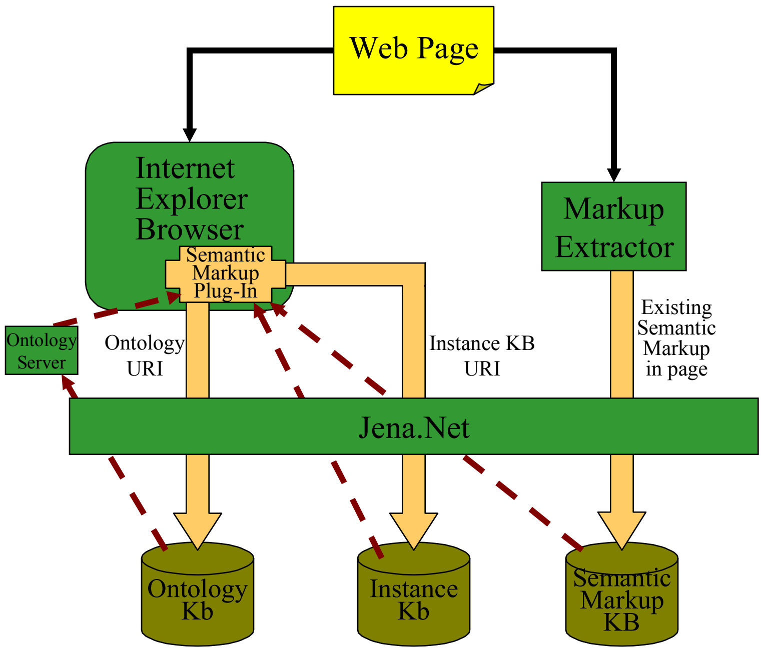


Figure 14. Semantic Markup Plug-In Architecture

We have a related project that is developing a semantic markup plug-in for Microsoft Word (in Microsoft Office these plug-ins are called add-ins) and much of the infrastructure developed for that project has been incorporated into our Browser Plug-In. In particular, the ActiveX control that displays the triples in a Semantic Markup Table, the Markup Chooser, and an Ontology Server have all been reused from that project.

The Ontology Server is an extension to Jena.Net that simplifies the extraction of ontological relations (e.g. direct sub- and super-types, and concept and property subsumption) and constraints (e.g. domain and range restrictions for properties) contained in the Ontology KB. It is used by the Markup Chooser to allow the user to navigate through the ontology's type hierarchy and to prune the listing of possible choices for a cell within a triple.

5. COMMON API FOR MANUAL MARKUP PLUGINS

Our MS Word and Internet Explorer plugins for interactive manual markup now share a common API for loading and querying both ontologies and models based on those ontologies. This API is implemented in terms of the Jena .Net library, BBN Technologies' port of the Java-based Jena library.

Both Jena and Jena .Net required each model instance to independently load the ontology(ies) whose terms that model uses. This resulted in very bad performance for applications in which there might be multiple small models based on the same, possibly large, ontology. We altered a few strategic classes in the Jena .Net implementation to remove this restriction. It is now possible for a model to indicate that its terminology is stored in a *different* model. These changes are upwards-compatible, and were communicated back to BBN.

6. ONTOLOGY CLASS LIBRARY GENERATOR

The .Net conversion mentioned before provided the basis for a new opportunity that we have pursued – the automatic generation of O-O programming libraries (.Net) from ontologies. Applications that read/write models based on an ontology are far easier to program through the classes of these libraries than through the classes of ontology-neutral libraries such as Jena.

Some semantic web applications are *generic* in nature – independent of any particular ontology. Validity checkers and query servers [1] fall into this category. Many more applications – e.g., a personal agent to manage bidding for an item at an online auction site -- require software that is written to deal with classes and properties of a specific ontology.

The current generation of ontology based programming tools provides only weak support for programming the ontology-specific class of applications. Better support could be provided by *automatic generation* of software class libraries based on the content of the ontologies needed for such an application.

The objective of this technique is to support implementation of ontology-specific applications as depicted in Figure 15, by automating the generation of ontology-specific class libraries for traditional object-oriented programming languages.

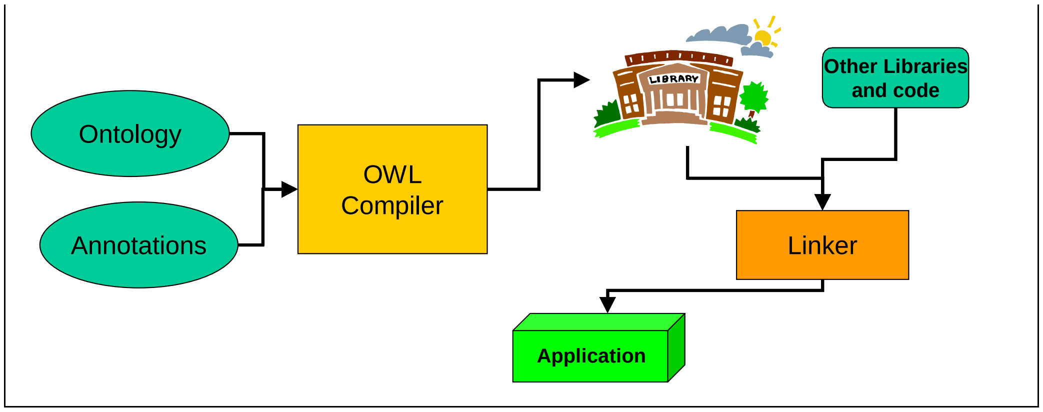


Figure 15. Ontology-specific Application Development.

An *ontology* and annotations are *compiled* to produce an ontology-specific *library*. This library is *linked* with other libraries and code to produce an *application*

6.1 Ontolog-Oriented (ON-O) vs. Object Oriented (OB-O) Data Modeling

Ontology languages, such as *OWL*, and object-oriented languages such as Microsoft's *.Net Framework* characterize a domain of discourse in quite similar fashion. In both we find a fundamental partitioning of the domain into *individuals* (**objects**) and *literals* (**values**). The universe of *individuals* (**objects**) is categorized into *classes* (**reference types**). The universe of *literals* (**values**) is categorized into *datatypes* (**value types**). OWL provides *ObjectProperties* to relate pairs of individuals, and *DatatypeProperties* to relate individuals to literals. .Net provides **fields**3 of **objects** for both purposes, but each field is used uniformly to relate an object to *either* another object *or* to a value, and so we could accurately partition them into **ObjectFields** and **DatatypeFields**.

Both *ObjectProperties* and *DatatypeProperties* can be given *domains* and *ranges*, restricting the individuals and literals that may be related by the properties. Every **field** also has a domain – some reference type – and a range.

Finally, both the On-O and Ob-O modeling languages provide intensional requirements of subsumption and disjointness4 – that is, statements or implications of subsumption and disjointness that must hold for *all* model instances. The Ob-O reference types include both **class types** and **interface types**. It is the interface types that bear a considerable similarity to the *classes* of On-O models – in particular, if I_1 and I_2 are interface types, then I_1 may "inherit"[5] from I_2 (and thus be subsumed by it). Alternatively, if I_1 and I_2 have no common subtype, then they must be disjoint.

[3] Software engineering concerns – primarily encapsulation – lead Ob-O languages to provide **methods** (and, in .Net, **properties**) in addition to Fields. However, **fields** capture the fundamental modeling of relationships in the domain.
[4] *Owl Lite* does not provide a way to declare classes to be disjoint, but other *Owl* variants and description logic languages do provide this.
[5] "Inherit" in this context is the imposition of a requirement that any class that implements I_1 must also implement I_2.

Another area of common concern is characterizing the number of *individuals* (**objects**) that may be related to a given *individual* (**object**) by a particular *ObjectProperty* (**field**). In particular, both On-O and Ob-O paradigms encourage a modeler to distinguish functional relationships (at most one range individual for a given domain individual) from others. The On-O paradigm cleanly isolates this issue, providing **restrictions** that allow cardinality restrictions other than "at most one" to be stated. The Ob-O paradigm confounds the characterization of cardinality with that of range. When a **field** is declared to have a reference type (e.g., Person) as its range, this means "at most one Person".[6] A programmer accommodates multiple range values by specifying a collection type (e.g., "array of Person") as the **field**'s range.[7]

The final commonality worth mentioning is that all the declarative information about classes (**reference types**), *datatypes* (**value types**), and *properties* (**fields**) in both paradigms is available in models of their own. This facility (referred to as **reflection** on Ob-O languages) is the basis for writing generic applications.

Although the commonalities are considerable, the Ob-O and On-O modeling paradigms are not isomorphs of one another. Each provides means of saying some things that the other cannot. These differences reflect the ways in which these two paradigms have been traditionally used. In particular, On-O models evolved with the expectation that they would be consumed by software that supplies sophisticated (relative to traditional software, at least) *inference* capabilities. Ideally, such software makes no distinction, other than performance, between a model where a statement is explicit and one where the statement not explicit but is a logical consequence of other statements and the ontology. For example, if the *ObjectProperty* "hasNationality" is stated to have the *Class* "Country" as its range, and the statement "x hasNationality y" appears in a model, then the individual y has "Country" as one of its classes in the model, regardless of whether the fact "y hasClass Country" was ever explicitly added to the model. Designers of On-O modeling languages like *Owl* carefully limit the logical expressiveness of the language so that such expectations are plausible.

Ob-O modelers, on the other hand, do not expect much inference to be performed by the automatically generated software that implements their models. They do expect the software to enforce the logical implications of their data modeling, but to do by restricting what models can be built. In particular, models are built incrementally and an increment is not allowed if the result would be inconsistent. In the Ob-O world, the increment "x hasNationality y" would be rejected unless "y hasClass Country" were already present. One significant exception to this is supertype inference. The Ob-O modeler *does* expect "y hasClass PoliticalUnit" to be inferred whenever "y hasClass Country" is explicit (or inferable), assuming Country was declared to be a subtype of PoliticalUnit. Ob-O programming languages like Java and the .Net family actually exclude inconsistent models primarily through static typing, rejecting any program that could build an inconsistent model,

[6] Ob-O languages traditionally support a Null object that can be stored in any reference-typed **field**. With very rare exceptions, use of the Null value is the means of saying "no range values for this domain object". There is no analog to the Null value for value types, so use of a value type as a **field** range means "exactly one."
[7] These collection types commit to more than just "multiple objects". They often commit to an ordering of the objects, or to allowing a bag rather than just a set.

rather than through runtime consistency checks. These languages provide *safe casts* to deal with circumstances where static type reasoning is too weak to ensure compliance. Programs that use reflection to build models are also subject to runtime exceptions from increments that would lead to inconsistent models.

The point of this discussion is that the On-O and Ob-O paradigms differ little in how they characterize a domain of discourse, but differ significantly in the anticipated use of that characterization by software. The one fundamental difference originates in the means by which *individuals* (**objects**) are categorized into *classes* (**reference types**). In the On-O paradigm, an individual may be explicitly (or implicitly) categorized into multiple classes. This *might* create an inconsistent model – e.g., if a pair of these classes had been declared disjoint in the ontology – but in general it is not a problem. In the Ob-O paradigm, however, each object belongs to a *single* most-specific, named, **class type**, and of course to any **class** or **interface** types that subsume it. As a consequence, the Ob-O data modeler must *anticipate* all possible intersections of types that may be needed by applications.

A second difference we shall have to confront in generating a class library suitable for manipulating models based on an ontology is the notion of identity. Owl has adopted from RDF the practice of using a URI as a globally shared "name" for an individual. While it is presumed that all uses of such a name refer to the same individual, it is *not* presumed that distinct names refer to distinct individuals. A given ontology and model will often justify neither the conclusion that the names refer to the same individual nor or that they refer to distinct individuals. In languages used to manipulate Ob-O models an individual is added to a model by applying a **New** operator, and each such individual is distinct, insofar as the language's innate reference equality semantics is concerned, from every other individual. As a result, an Ob-O model cannot contain both "a person named Will Shakespeare" and "a person named Francis Bacon" without making a commitment as to whether they are the same or distinct individuals. On-O models, on the other hand, have the option of stating that they are the same, stating that they are different, or remaining noncommittal. [8]

6.2 Generic vs. Ontology-Specific Programming

If we accept that the primary value of the Semantic Web will come from software applications that can create and reason about ontology-based models, we need to ask whether existing programming languages are well-suited to building these applications.

The kinds of applications that are generally suggested for the Semantic Web seem to fall into two very different camps. Some applications – perhaps better thought of as tools – are generic in that their construction requires knowledge of the meaning of terms used in creating ontologies, such as 'subClassOf', but requires no knowledge of terms or semantics from any specific ontology. The kinds of tasks performed by such tools are things like consistency checking and query answering.

[8] Typically Ob-O languages predefine a coarser-grained *Equals* equivalence operation that can be overridden on a class-by-class basis. But *Equals*, like reference equality, is boolean-valued; it does not provide "maybe" as a possible result.

A second kind of application is one that deals with models based on one or more specific ontologies posits an agent that finds a suitable physician for a patient. This agent is concerned with some medical terms and some distance and time concepts. There are two reasons why non-generic applications are necessary:

Some agents will need to supply semantics that cannot be expressed in the ontology language. Recall that ontology languages purposely limit their expressivity to retain decidability. The day may come when the additional semantics can be expressed in a standardized "declarative" language; it is possible today to supply them with procedural code.

Even where the semantics expressed in an ontology are *logically* sufficient for an application, performance considerations will often dictate a non-generic implementation.

One approach to building Semantic Web tools and applications that has been popular is to rely on class libraries for mainstream object oriented languages (Java, .Net). The class library provides classes and fields that hold both ontologies and models built from them. Import/export methods provide a means to map between this representation and an XML-based "interchange" representation. The library provides two additional capabilities:

a limited amount of inference, following the semantics expressed by the loaded ontology. Which of the authorized inferences are provided seems to be an ad-hoc implementation decision.

a reflection-like set of methods that allow a programmer to index into a model using the types and properties of the ontologies on which it is based.

Figure 16 exhibits the use of such a library to write a fragment of the sort of code that would be used in an ontology-specific application. The code uses the *Jena* class library for Java, and is taken directly from a tutorial provided by the authors of that library. Jena is a library that can manipulate pure RDF models as well as models that use the OWL extensions. The encoded algorithm is examining a model based on the VCARD ontology[9], attempting to determine the name of a person whose email address is "amanda_cartwright@example.org".

[9]http://www.hpl.hp.com/semweb/iswc2002/JenaTutorial.Alpha/DAML/solutions/vcard-daml.rdf

```
DAMLModel model = … // code that loads the VCARD ontology
                    // and some data based on that ontology
DAMLClass vcardClass =
    (DAMLClass)model.getDAMLValue(vcardBaseURI+"#VCARD");
DAMLProperty fnProp =
   (DAMLProperty) model.getDAMLValue(vcardBaseURI+"#FN");
DAMLProperty emailProp = (DAMLProperty)
                 model.getDAMLValue(vcardBaseURI+"#EMAIL");
Iterator i = vcardClass.getInstances();
while (i.hasNext()) {
  DAMLInstance vcard = (DAMLInstance) i.next();
  Iterator i2 =
      vcard.accessProperty(emailProp).getAll(true);
  while (i2.hasNext()) {
     DAMLInstance email = (DAMLInstance) i2.next();
     if (email.getProperty(RDF.value).getString().equals(
                  "amanda_cartwright@example.org" ) ) {
       DAMLDataInstance fullname =
        (DAMLDataInstance) vcard.accessProperty(fnProp).
                           getDAMLValue();
       if ( fullname != null )
        System.out.println("Name: "+ fullname.getValue().
                                  getString());
     }
  }
}
```

Figure 16. Jena code used for an ontology specific application

Several points stand out in an examination of this code.

The Java types of the program's variables (**DAMLModel**, **DAMLClass**, **DAMLProperty**, **DAMLInstance**, **DAMLDataInstance**) are analogous to reflection types. They are *not* the classes introduced by the VCARD ontology. They *are* classes a programmer would want for writing generic applications.

The names of classes and properties from the VCARD ontology appear as string literals (*VCARD*, *FN*, *EMAIL*).

Datatype values stored in the model must be explicitly converted (getString) to an appropriate Java value type.

Figure 17 displays the same algorithm, as it would appear in an object-oriented language using a VCARD class library. This example uses the syntax of VisualBasic.Net, but the code would isomorphic or nearly so in C# or even Java. In this case:

The variable types (**VCARD, String**) and method names (*vcards*, *email*, *fn*) are those introduced or used by the VCARD ontology itself, not reflection types.

Values of datatypes (in this example, strings) appear to the programmer as values from the value types of the embedding language.[10]

[10] Technically "string", unlike numeric types, is a reference type in the .Net languages as well as in Java. However, since strings have no mutable state, and provide a value-based equality operator, "string" acts for all practical purposes as an optimized value type.

```
Dim model as VcardOntology.model
Dim vc as Vcard
for each vc in model.vcards
  dim eaddr as String
  for each eaddr in vc.emails
    if eaddr.equals("amanda_cartwright@example.org") then
      dim fullname as String = vc.fn
      if not fullname is Nothing then
         Console.Writeline("Name: " & fullname)
      end if
    end if
  next eaddr
next vc
```

Figure 17. Statically typed code for an ontology specific application

The code in Figure 17 is objectively more concise, and subjectively easier to comprehend, than that of Figure 16. Programming in a statically typed environment conveys other important benefits. First, numerous programming errors that would only be detected by debugging in the Jena code will be caught by the compiler in the .Net code. These range from simple typographical errors, such as using "#*VCRD*" or "#vcard" instead of "#VCARD", to semantic errors that show up as type mismatches, such as using "#Orgname", a property whose domain is **Orgproperties**, rather than "VCard", in place of "#FN", a property whose domain is **Vcard**. In fact, such errors are often detected prior to compilation, or avoided entirely, in modern IDEs for statically typed languages, since programmers are offered menus and automatic name completion restricted to type-correct choices. Finally, the compilers of statically typed languages take advantage of the statically declared data model to produce smaller and more efficient code.

So far we have only observed that a *programmer* could take the same domain expertise required to write an ontology and could himself write an Ob-O class library that was more suitable for implementing many, if not all, ontology-specific applications for that domain. This is not surprising. A well-written ontology already contains a declarative representation of the knowledge needed to construct that library – i.e., the library can be *generated* from the ontology automatically. The generated library can easily retain the correspondence between its types and fields and the corresponding concepts in the ontology as a mapping to the concept's URIs. Making this mapping available to an Ob-O language's reflection facilities enables a single *generic* program to import existing models encoded in their interchange syntax (RDF/XML) into the class library representation, and to export models created within the class library in interchange syntax. Thus no additional code needs to be written, or even generated, for the class libraries to share data with other web agents.

6.3 Not so fast…

Just what is the relationship between the Jena code in Figure 15 and the .Net code in Figure 17? Will they produce the same results across all possible model instances? Jena, after all, is capable of storing even inconsistent models. Furthermore, these code snippets contain no code that actually builds or increments models. Can the .Net statically typed library construct all possible models that could be constructed by the Jena code?

The answers to these questions depend on the issues of consistency and inference alluded to the previous section. A model, consisting of a finite set of assertions P(d,r) is *inconsistent* only if the negation of one of those assertions can be *inferred* from the model and the ontology. Since negations cannot be explicitly asserted, this must arise from an inferable negation. In the case of the Owl family this will involve at least one of the following:

- Asserting that two individuals are distinct
- Declaring that a property is functional (enabling inference of distinctness)
- Declaring that two classes are disjoint
- Declaring that one class is the complement of another (entailing their disjointness)

Many useful ontology specific applications do not rely on the ability to represent inconsistent models. Other suggested applications – particularly those that attempt to bring together data about the same domain of discourse from disparate sources – do seem likely to need to hold inconsistent models.[11]

At first blush the notion of static typing may seem incompatible with the ability to hold inconsistent models. In fact they are not incompatible. This will be addressed in greater detail in the following section. For now, simply observe that an Ob-O language can readily provide multiple methods that read and write the same data storage. By using run-time casts in their implementation, these methods can provide *different* views regarding the type of the storage. For example, what appears to be an "array of Thing" when viewed through some methods may appear as an "array of Person" when view through alternative methods. What allows this to work is that some of the casts used in the "array of Person" view will raise exceptions in the case that a non-Person has been added to the array through the "array of Thing" view. In summary, there is no reason why a class library cannot simultaneously provide a statically typed view of a model and simultaneously provide a statically typeless view of the same model. The typeless view would be capable, just like the Jena library, of holding an arbitrary set of explicit assertions.

Inference, on the other hand, is the ability to materialize assertions that are entailed by as well as those that are explicit in a model. Since any entailed assertion *could have been* explicitly asserted, inference does not introduce any added problems for the statically typed view of the model.

6.4 Generating an OB-O Class Library from an Ontology

The class library generator accepts as input an ontology (currently, DAML+OIL or OWL) and produces as output a .Net class library. It produces this library both in its binary form (an assembly) and in source code form (both VisualBasic and C#). Production of multiple source languages is handled by .Net's object model for code, which allows code generators to create abstract program models which can be both compiled and translated to the various concrete .Net languages.

[11] In this regard, it is of concern that the interfaces through which people author ontologies and models to date seem to closely parallel the underlying language primitives, making it likely that desirable assertions of distinctness and declarations of disjointness will be inadvertently omitted.

The scheme by which this generator works was outlined above. The strategy is to provide pairs of typed and untyped fields for storage. Each typed field is associated with a .Net type T and holds a list of references each of which is a .Net instance of T – that is, **`TypeOf r is T`** would be true for every reference r in the list. T is a type in the generated library that serves as the "implementation" of a class in the ontology. The paired untyped field will hold references that need to be treated as instances of that type, but whose .Net type is not consistent with it. We will soon see where such instances arise. Corresponding to each such field pair, the library will provide one set of properties and methods that provide a statically typed view of the model and raise an exception if the untyped field is non-empty. A second set of properties and methods provides a typeless view that treats the concatenation of the pair of fields as a single collection of "Thing"s. The typeless view supports (portions of) applications that cannot abide the requirements of static typing.

In generating the types for a class library's fields, properties, and methods only *global* domain and range restrictions are taken into account. Implications about domain class that are conditional on range class and vice versa cannot be enforced statically and thus play no role in the assignment.

The descriptions below will refer to the accessibility of fields, methods, and properties as simply Public or Private. However, in many cases what is listed as Private actually uses the accessibility scope (called *Friend* in VisualBasic or i*nternal* in C#) that allows an element to be accessed from elsewhere in the library containing the element but not from outside that library.

6.4.1 Mapping Individual Classes

Suppose the ontology is named O. The class library will contain a public class **O.Model**. For each non-anonymous class C in O, the class library will contain a private class **O.C**. **O.C** will have a zero-argument constructor. The class **O.Model** will have a private field **my_Cs**, of type ArrayList[12]. This will hold only objects created by the constructor of **O.C**. The initial value is the empty list. A public method **O.Model.NewC()** implements allocation of new instances of **O.C**, calling its constructor and adding the result to **my_Cs**.

The library will also have a public interface **O.IC.** **O.C** will implement **O.IC.** We need the interface, as well as the class, because *multiple* inheritance only exists for interface types, not class types.

6.4.2 Mapping Subclassof

Suppose that the closest non-anonymous superclasses of C in O are $\{Sup_1 \ldots Sup_j\}$. Then for each $1 \leq i \leq j$, we will have **O.IC** inherits **O.ISup$_i$**. The net effect is that **O.C** now

[12] This is an artifact of the absence of templates in the .Net languages. The only directly supported statically typed collection of Xs is "X()" (array of X), but these are not growable like ArrayLists. Thus we use a more flexible collection type and pay the price of a run-time cast (in the library code) each time an item is obtained from the list.

implements its direct interface, **O.IC**, which in turn inherits each interfaces paired with a superclass. Since this pattern is followed for *every* class, **O.C** is forced to implement the interface of every class in its superclass hierarchy. Figure 18 depicts this mapping in a simple case.

Suppose also that the closest non-anonymous subclasses of C in O are $\{Sub_1 \ldots Sub_n\}$. There will be a public method **O.Model.EnumerateCs** that returns a statically typed enumerator of **O.C**s. This enumeration will include objects in **my_Cs**, as well as objects produced by **EnumerateSub$_1$s**, etc. In other words, it will include objects created by the constructor of **O.C** or by the constructor of any of the classes corresponding to subclasses of C. In the case of DAGs, of course, this enumerator does *not* enumerate a class multiple times.

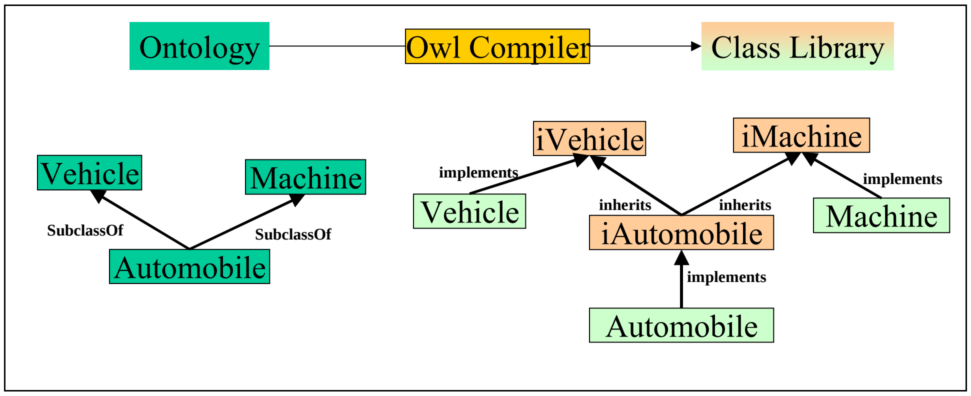


Figure 18. Ontology Classes

For each ontology class, the generated library contains both a *class* and an *interface*. *SubclassOf* relationships between ontology classes are mirrored by *inherits* relationships among interfaces. Each library class *implements* the interface with which it is paired.

6.4.3 Mapping ObjectProperty and DatatypeProperty

Let $\mathcal{P}$(C) be the set of ObjectProperties in O whose global domain is declared to be C or a (not necessarily direct) superclass of C. If p is an ObjectProperty with no global domain declaration, then the domain is Thing, and we include p in $\mathcal{P}$(C).[13] In other words, $\mathcal{P}$(C) contains those properties applicable to *all* instances of C. For each $p \in \mathcal{P}(C)$, we give **O.C** a private field **O.C.myps**. The field is assigned the type ArrayList, and initialized to the empty list. This provides a place to store the individuals whose .Net type is compatible with the declared range of p. A second field, **O.C.myps_nost**, is used to store other values for the range. Of course, if there is no declared range, or the range is Thing, the latter variable will always hold the empty list.

[13] If p has multiple global domain declarations, then its domain is their intersection. We invent a name for this class and proceed as if the intersection class had been declared explicitly in the ontology and used as the domain of p.

Let $\mathcal{P}'$(C) be the set of ObjectProperties in O whose global domain is declared to be C itself. For each p in $\mathcal{P}'$(C), **O.IC** is given two methods for adding a new range value for p. In one, **Addp**, the parameter type is the interface type corresponding to the range of p. In the other, **Addp_nost**, the parameter type is Thing. The implementations of the former simply add the parameter to **myps**. The implementations for the latter perform a run-time type test and add the parameter to **myps** if the reference has the right type, and to **myps_nost** if not.

If p has a global maximum range cardinality of 1, then an additional pair of writing methods is included in **O.IC**. These (**Setp, Setp_nost)** provide a value that becomes the *sole* value in **myps** or **myps_nost.**

For reading, both statically typed and untyped iteration are provided. The statically typed method enumerates values from **myps**, but raises an exception if **myps_nost** is non-empty. The untyped iteration enumerates references from both lists.

If p has a global maxcardinality of 1, an additional statically typed reading method is provided. This method is prototyped as `GetP(byval ifnone as O.IR)as O.IR` where R is global range restriction for p. The implementation consists of the following logic:

If **mypsnost** is nonempty, an exception is raised. Otherwise

If **myps** holds exactly one reference, that reference is returned. Otherwise

If **myps** holds no values, the "ifnone" parameter is returned. Otherwise

If **myps** holds multiple values, an exception is raised.

If p also has a global mincardinality of 1, an additional method is prototyped as `GetP()as O.IR` and functions identically to the other **GetP** method, except in case (3), where the second method raises an exception. An implementer's choice among these various methods reflects a decision on whether to treat inconsistency or incompleteness of a model as an *exceptional* situation or as a *normal* circumstance.

The mapping for Datatype properties is completely analogous. Because the fields that store range values always contain lists, regardless of any cardinality restrictions, the absence of a Null value for these types does not pose a problem.

6.4.4 What is New and What New is not

We have just seen the how the class library increments a model with a single triple involving a property from O. But there are a few other kinds of triples that get added to Jena models, and we should look at their counterparts as well. Most interesting is the analog of the triple Jena uses to say "individual I belongs to class C." For Jena, there is nothing special about the ObjectProperty for "Type". But a statically typed language will somewhere have to treat this as other than just another property. To understand why there is a problem, we revisit the **New** operator mentioned earlier in conjunction with the Ob-O notion of identity. The New operator does more than give a new object an identity. It also gives the object a classification, which must be a single class (not interface) type C. The object returned by this operator is known to the Ob-O runtime to be an instance of that class, to be an instance of the class from which C inherits, etc, back to the built-in class **Object**, which is the root of all class inheritance. Furthermore, the runtime knows that if the object is an instance of any class type C, it is also an instance of the type I for every interface I implemented by C.

But while the **New** operator does a lot of classification, it will not do "additional" classification or "reclassification" or "specialization". If s is a variable bound to a Swimmer, we cannot write s.New.Bowler() to make s be a Bowler as well, or write s.New.NavySeal() to make s a particularly good swimmer. **New** just isn't used like a method; it is a syntactic primitive. So how do Ob-O languages perform these model increments, so straightforward in Jena and its ilk? As pointed out ina previous section, they *don't*. A tradeoff has been made in the runtime of these languages to favor performance over this flexibility. Our ontology compiler, in order to regain that flexibility, sacrifices performance.

For any object returned by **Model.NewC()**, we allow applications to create "proxies" of other types at any time. For example, having executed

ISwimmer s = m.NewSwimmer()

we can later execute

IBowler b = m.AsBowler(s)

The object created and stored in b is created when AsBowler uses m.NewBowler. The Bowler instance gets added to the **O.Model.my_Bowlers** list. We call the Bowler a *proxy* for the Swimmer instance, and we call the Swimmer a *principal* object of m and *the principal* object for the bowler instance. This information is recorded in private fields of the two instances. We also say that the Swimmer is its own principal object. Either the swimmer or the bowler may be used as the parameter of `m.AsSumoWrestler` to create another proxy. A principal object may end up with proxies of many other types, but at most one per type. If s had already had a proxy of the exact class bowler when we executed `m.AsBowler(s)`, the existing proxy would have been returned. Furthermore, a principal may not have a proxy of a class that is the same as or a superclass of the principal's own class. Having *created* s as a Swimmer, attempting to classify s as an athlete would have no affect on the model -- `m.AsAthlete(s)` would just return the Swimmer s.

The previous paragraph depicts a mechanism that in principal *stores* multiple classifications for a reference. We must also provide alternatives for the .Net runtime's operations that we wish were aware of this. In particular, applications that need proxies also need

an equality test that treats all objects having the same principal as equivalent,

a test of whether an object x is an instance of a type T that returns true if any object y having the same principal as x is an instance of T in the .Net sense, and

a casting operator that will successfully cast x to T if any object having the same principal as x could be cast to T by a .Net cast.

The first method is a straightforward overloading of *Equals*. Since types are first-class objects in the .Net reflection API, the latter two are likewise easy to implement.

Note that this is *not* a matter of implementing inferences authorized by a model and ontology. The swimmer and bowler did not arise from descriptions with distinct ids. They fail the innate identity test solely because, to satisfy the requirements of static typing, two different invocations of the New operator were needed.

6.4.5 Retaining the URI mapping

The .Net languages contain a weak but still useful means of extending the reflection API. By decorating the declarations of classes, interfaces, methods, etc. with some optional syntax, the .Net compiler arranges for the runtime to create objects called *custom attributes* whenever the library is loaded and associate these objects with the reflection instance corresponding to the decorated unit. There is very little restriction on what can go into classes used as custom attributes. So generating the code

```
    <Resource("http://www.w3.org/2001/vcard-rdf/3.0#VCARD”>                 _
Public Interface IVCARD  …    End Interface
```

suffices to make the URI used for the class VCARD available via reflection to any application that loads the generated library.

6.5 Annotations

Ontologies do not address some issues that a programmer considers when designing an Ob-O class library. These are issues that rely on knowledge (or assumptions) about the applications that will actually be implemented using the library.

The four primary considerations that programmers bring to bear in these considerations are code size, data size, performance, and encapsulation. The latter is particularly important to Ob-O programmers, since it is central to their ability to provide structured explanations of algorithm correctness and security.

The scheme outlined in section 0 produces a library that is larger, slower, and less encapsulated than one that a programmer would likely produce manually. It will, on the other hand, be more flexible – that is, supply the functionality needed for a wider variety of possible applications – than the typical manually produced library. We now consider some *annotations* that can be asserted about the classes and properties of an ontology to control these tradeoffs. The list is meant only to be indicative of what is possible, not an exhaustive enumeration of what would be useful.

Class libraries do not always include a root object like *model* whose instances effectively partition the instances of all other objects. Even when they do, a root can generally enumerate the instances of few, if any, reference types in the library. A programmer can authorize the generator to omit the enumeration method for a type by marking the type as non-enumerable. A non-enumerable type may not have an enumerable supertype.

Ob-O programmers are comfortable with explicitly classifying an individual prior to using that individual in a context that requires (implies) its membership in the class -- they must circumvent the static typing system when this is not the case. Most programmers would argue that the value of trapping unintentional errors in the static type checker far outweighs the occasional inconvenience of having to provide explicit classifications. For most properties, a programmer would be happy to give up the flexibility of adding assertions about individuals not statically known to belong to the declared domain/range class. In other words, they are willing to allow the global domain and range declarations for a property be treated as static prerequisites for explicit assertions.

The .Net languages (and Java) allow a class not only to implement multiple interfaces, but to inherit the implementation of one other class. This means that it shares the implementation of any interfaces it has in common with that class, including fields. This has no impact on functionality or flexibility, but can result in drastically smaller code. Using annotations to indicate a single-inheritance thread through some of the classes in an ontology would significantly reduce the amount of code duplication needed to implement interfaces in the library's classes.

Many types in .Net programs exist *only* to be used as an interfaces – no class is needed that implements just that one interface. For example, an annotation could be used to indicate that "Vehicle" did not need to be supported as a class with its own implementation, but only as an interface implemented by its subclasses.

Classes in .Net programs can be marked as "MustInherit", meaning that any instance of the class must actually be created by instantiating some class that inherits from it. A class defined in an ontology as the union of other classes would be a likely candidate for this annotation.

Table 1. Annotation Tradeoffs

Annotation	Code Size	Data Size	Performance	Encapsulation
Enumerability	☺	☺	☺	
Domain/range as static requirements	☺	☺		☺
InheritsFrom	☺			
InterfaceOnly	☺			☺
MustInherit				☺
Bitmask		☺	☺	

Enumerated types in programming languages are generally implemented by assigning, at compile time, a distinct integer value to each enumeration member and using the integers as the run-time representation of the member. In .Net, programmers can choose these integers, and a common programming technique is to choose integers whose binary representation can be effectively used as a bit mask, thus enabling some logical set operations to be performed in constant time. This optimization could be expressed through annotations.

Table 1 above summarizes the tradeoffs inherent in each of these annotations. The choice of the "default" flexibility to provide in a generated class library could be one that provided great flexibility and required annotations to give up some of that flexibility. Alternatively, the default could be for a relatively lean but inflexible library, requiring annotations to include the additional flexibility.

6.6 Discussion

A number of concerns for generating a viable class library implementation were omitted entirely from the design sketched in the previous sections. I will touch on several of them briefly here.

The generated code must supply *names* for classes, interfaces, properties, methods, and fields. The names must be legal, must avoid collisions with one another and with names in libraries on which the generated class library depends. The public names should be recognizable by a programmer who is familiar with the ontology being implemented. None of this is a serious problem. In both .Net and OWL ontologies, names consist of a namespace and a local part. Keeping a 1-1 correspondence between namespaces avoids most potential collisions, and relying on the local portion of the names from the ontology as the basis for the local portion of the class library names resolves the latter problem.

The discussion omitted operations that *remove* facts from a model. For the most part, those are just the obvious complements of the operations that add the same facts. The one exception is the classifications done by **New**, which inherently have the same lifetime as the object created. One could attempt to hide those classifications from the operations implemented by the library itself, but they would still be visible to the OO runtime system and so could not be effectively hidden from applications.

Scalability is a serious issue. The design presented here views the ontology to be implemented as a self-contained unit. Clearly we want to be able to reuse (share) the binary implementation of one ontology in the implementation of ontologies that import it. But there don't appear to be any restrictions in OWL on how an ontology that imports another one can use the terms of the imported one. The importing ontology may add new superclasses to the classes in the imported ontology, add new properties using classes of the imported ontology as their domain, or add global range requirements to imported properties. This is all very unlike the way class libraries use and extend one another.

The library generator description in this document largely ignored *Datatypes and DatatypeProperties*. They are much simpler to deal with than Objects and ObjectProperties for three reasons. First, there is no possibility of the equality of two datatype values being uncommitted. Second, there is no possibility of *asserting* the type of a datatype value. Third, there doesn't seem to be any practical need to store a datatype value in the range of a DatatypeProperty when the value is the wrong type – it seems that raising an exception in the case where there is no obvious conversion would be perfectly acceptable. It is also hard to imagine a use for a DatatypeProperty with no range type specified. The remaining problem is matching up the XSD types with the .Net value types, and the correspondence is straightforward for all the commonly used types, with the exception of fact that XSD integers have no size limit. Transparently mapping between an ontology's datatypes and built-in types opens up to the application programmer the large body of predefined methods on those types.

The use of proxies and principals is inadequate, as described, to fully deal with the problem of multiple explicit types for a single individual. Consider the example used earlier. Any property declared with Athlete as its domain would have ended up with explicit storage in both the principal, Swimmer, and in the proxy, Bowler, instances. If IsProfessional were such a property, we would have two values around. Of course we know it is possible to be both a professional basketball player and an amateur at baseball, so the same might be true for swimming and bowling. But this just shows the dangers of thinking you can understand the meaning of a property from its name – clearly the semantics of OWL requires that if IsProfessional is a property of athletes, there is no sense in which multiple values of the property can be assigned to an individual and differentiated based on a subclass when the individual happens to belong to more than one subclass.

6.7 Implementation

6.7.1 Processing Owl from .Net

Currently libraries for processing Owl models are only available in Java. Rather than expend significant resources building .Net based internalizer/reasoner for OWL (e.g., porting Jena2) we explored two alternatives:

A commercial software bridge that would allow .Net code to call methods from Java class libraries.

An ad-hoc (e.g., socket) protocol to communicate between .Net code and Java code.

We determined that, until the set of queries that the library generator needs to pose to the Java is stable, it was inappropriate to use an ad-hoc protocol. We chose a commercial product, JNBridge™, which gives us a software bridge between .Net and Java code. The overhead for each method invocation is very high, because JNBridge™ requires the Java JVM to execute in a distinct process from the .Net application, but we have reduced the impact to acceptable levels for our application through a few additional Java methods allowing multiple results to be returned in a single method call:

Jena2 library queries generally return collections as an Iterator. Use of an Iterator requires two method calls per element (hasNext, getNext). In almost all cases, our .Net client needs to consume *all* the values in a collection. So we invoke one additional Java method to copy the elements produced by the Iterator into an array. This additional method returns the array *by value* to the .Net application. So only 2 bridge calls, rather than 2N, are needed to obtain all N values from an Iterator. The added overhead of building and transmitting an array by value is negligible compared to the bridge call overhead.

Most results we need from the Jena2 code are either *Resource*s or collections of (Iterators over) *Resource*s. When we obtain a resource, we nearly always need to obtain its *URI*. This requires an additional Jena2 method call. We added methods to return, *by value*, *PackagedResource*s, which are just structures with two fields, a *Resource* and a *String*(the *URI* of the resource). This eliminates a large number of extra method calls.

We also implemented an interface to the JNBridge™ runtime that allows the JVM to run in a separate thread of the .Net application, rather than in a distinct process. This cuts the per call overhead slightly (but less than 10%).

The Class Library Generator was deployed as a free web-based service on http://mr.teknowledge.com. A user of the web service simply fills in a form with the *url* of an ontology, selects the ontology language (DAML or one of the OWL species), and submits the form. The returned page contains a link to a zip archive containing both the generated library source code (in VisualBasic or CSharp, at the submitter's discretion) and the binary .Net assembly for the library. The submitter does not need any additional software on his machine, nor does he need a .Net/Java bridge to make use of the generated library.

An alternative commercial bridge, JuggerNet, allows calls from .Net to Java to occur without a thread switch, and so should provide much better performance for our application. However, we felt that the licensing cost of this product was prohibitive.

We retrofitted the Briefing Associate so that it can use a class library generated directly from an ontology both as its exchanger for that ontology and as the basis for its visual annotation GUI.

7. DELIVERABLES

Our software deliverables are not web-based tools, but plugins for commercial desktop applications (MS PowerPoint, MS Word). Both are currently deployed in the form of Windows NT/2000/XP installation modules (.msi files). Downloadable copies exist on the Teknowledge web server.

In order to help users familiarize themselves with the Briefing Associate, we also published on the website a tutorial covering the following aspects of its use:

- Ontology creation
- Ontology annotation
- Creation of marked-up briefings through GUI extensions
- Implementation and invocation of ontology-specific briefing analyses
- Markup publication

8. PRESENTATIONS AND PUBLICATIONS

8.1 Intelligence Community Presentation

We installed a copy of the Briefing Associate software on a laptop that was used by in a presentation to the 2001 IntelLink conference. The presentation included a live demonstration of using the Briefing Associate to create markup as a byproduct of composing a briefing, and of the use of semantic analysis tools to detect flaws in a briefing's content.

Dr. Robert Balzer gave a presentation on the Briefing Associate at the IntelLink Briefing organized by Dr. Jim Hendler (DARPA DAML Program Manger) at the DARPA Technology Interchange Center (TIC). The presentation was one of four selected by Dr. Hendler to present the technical concept and potential applicability of the semantic web to technology representatives of intelligence agencies.

An initial version of template-based markup production within Microsoft Word specialized to the HORUS ontologies was transferred to a BBN Technologies' computer and used for a demonstration presented at the Semantic Web for the Military User conference in May 2003.

8.2 Conference Presentations

A paper describing the Briefing Associate was published in the proceedings of the 2001 International Semantic Web Working Symposium in Palo Alto, CA. Dr. Marcelo Tallis attended the symposium and presented the paper.

An article on the Ontology Compiler -- *Ontology-Oriented Programming: Static Typing for the Inconsistent Programmer* by Neil M. Goldman was published in Lecture Notes in Computer Science: The SemanticWeb - ISWC 2003, Springer-Verlag Heidelberg, Volume 2870 / 2003, September 2003, pp. 850 - 865. Dr. Neil M. Goldman attended the conference and presented the paper.

A paper on Semantic Word -- "Semantic Word Processing for Content Authors" by Marcelo Tallis was published in the Workshop Notes of the Knowledge Markup and Semantic Annotation Workshop (SEMANNOT 2003), Second International Conference on Knowledge Capture (K-CAP 2003), October 26, 2003, Sanibel, Florida, USA. Dr. Marcelo Tallis attended the workshop and presented the paper.

8.3 Journal Publication

An article on the Briefing Associate --The Briefing Associate: Easing Authors into the Semantic Web, by Tallis, Goldman, and Balzer, was published in the IEEE Intelligent Systems Journal (Vol. 17, no. 1).